AspectD: Enhancing a Standard Debugger with Aspects

Massimiliano Menarini
Department of Computer Science and Engineering, UC San Diego
9500 Gilman Drive, Mail Code 0404
La Jolla, CA 92093-0404, USA
+1 (858) 822-0424
mmenarini@ucsd.edu

Yan Yan
Department of Computer Science and Engineering, UC San Diego
9500 Gilman Drive, Mail Code 0404
La Jolla, CA 92093-0404, USA
+1 (858) 534-9912
yayan@cs.ucsd.edu

William G. Griswold
Department of Computer Science and Engineering, UC San Diego
9500 Gilman Drive, Mail Code 0404
La Jolla, CA 92093-0404, USA
+1 (858) 534-6898
wgg@cs.ucsd.edu

ABSTRACT
Aspect-oriented techniques have gained some favor in program tracing, profiling, and debugging. Yet, taking advantage of AOP typically requires giving up the traditional benefits of using a debugger, such as interactive exploratory introspection and remote execution. However, the APIs debuggers present to their scripting languages are not designed for aspect weaving, creating technical challenges. This paper discusses the programming primitives debuggers must support for weaving aspects into programs. The implementation choices in a debugger can lead to substantial differences in expressiveness and performance of the aspect-oriented language that can be built on top of the debugger. As an initial exploration of the design tradeoffs and implementation possibilities, we present AspectD, a tool that uses the Java Debug Interface (JDI) to weave simple aspects into Java programs. We analyze the performance and complexity of different types of advice in real-world program analysis and identify shortcomings of the current JDI API and propose improvements to enable the use of JDI for weaving aspects. The resulting lessons provide a design space for adding aspect orientation to a wide class of debuggers.

Categories and Subject Descriptors

General Terms
Performance, Design, Languages.

Keywords
Aspect-oriented language, debugging, runtime analysis

1. INTRODUCTION
Tools for program tracing, profiling, and debugging are the key to successful software development of complex systems. Aspect-oriented techniques have gained some favor for these activities, as it is possible to concisely identify a large number and variety of points of interest in the program by means of declarative pointcuts that specify at what points in an execution advice – logging or printing instructions, etc. – should be executed [1, 2]. Researchers have even used aspect-oriented techniques for runtime verification. In this context, pointcuts identified events of interest for the verification, and advice implemented state machines accepting the property that researchers wanted to verify at runtime [3].

Yet, taking advantage of AOP typically requires giving up many of the traditional benefits of using a debugger. In particular, the debugging (etc.) process with AOP is typically one of programming, that is, adding aspect code to the application. This results in several compromises:

- The often exploratory, iterative debugging process inherits an extra costly re-compilation step, and possibly redeployment to the execution setting.
- Complex applications often require remote monitoring, such as when they are deployed in their production environment, requiring additional, complex infrastructure.
- Aspects add code to the application to be debugged. This can perturb the application’s behavior (e.g., change memory use and layout), as well as require extra programming effort to keep the aspect code itself from being logged or profiled.
- Complex applications and frameworks, by their very nature, use reflection or on-the-fly code synthesis. For example, the enterprise Java framework WildFly [4] uses ASM [5] to manipulate an application’s Java byte code instead of using reflection to improve performance. Frameworks like Spring [6], which creates dynamic proxies for weaving and supports a subset of AspectJ [7] pointcuts, will miss ASM’s dynamically-introduced join points. Other AOP programming approaches, typified by PROSE [8], [9], use runtime compilation and weaving to accommodate this reality, but because they use the same methods as the applications they are weaving, there can be unintended interactions. We discuss these in greater detail in the next Section.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Conference’10, Month 1–2, 2010, City, State, Country.
Copyright 2010 ACM 1-58113-000-0/00/00010 …$15.00.
In this paper we investigate the capabilities of one popular debugger, the standard Java debugger, in supporting aspect-oriented debugging (AOD) and generate the requirements for any debugger to support AOD. We conduct this analysis by leveraging features available in most debuggers, such as the creation of conditional breakpoints and the modification of program variables or memory content, and use them in the design and implementation of AspectD, a dynamic weaving tool that supports a simple AOP language. The resulting analysis reveals that the Java debugger is missing key features. The omission of these features dramatically impact performance, but are not onerous to support. Indeed, some of these missing features would be useful to improving the performance of programmatic debugging in general, not just AOD.

AspectD shares many of the benefits of traditional aspect-oriented techniques but, instead of relying on the modification of the binary code, it uses conditional breakpoints to weave aspects. This technique presents novel challenges, but provides important benefits. Using debugger breakpoints we support a powerful pointcut language to monitor the execution of applications. This approach works nicely with complex applications that use reflection or aspects themselves. Debugger support for the modification of program state can be used to inject values into the running program or modify the program flow.

A simple example of an advice in our language is the following:

```java
@access:field.access:edu.ucsd.delphi.*; => spy(@access)
```

which instructs the debugger to run the `spy` method for every access to any field defined in the classes contained in the package `edu.ucsd.delphi`. This simple example is part of the case study presented in section 5.1. It implements a field access watch, a typical activity in debugging. Note that AspectD, using the debugger API, supports accessing even private fields.

Although it would be beneficial to support a pointcut language that allows any regular expression, we limit ourselves to the subset of regular expressions supported by the JDI API. This is one of the tradeoffs imposed by using a debugger to weave aspects. By relying on the primitives provided by the debugger we can instrument the code without much overhead. Implementing a more general regular expression language using lower level primitives in the debugger, while possible in theory, leads to a substantial performance penalty. This simple example shows how important the design of the debugger API is in enabling powerful and efficient pointcut definitions.

The contributions of this paper are several-fold:

- Requirements for a debugging system to support efficient and expressive AOD, without giving up the remote monitoring, run-time weaving, and separation of debugging code (aspects) from application (base code).
- The design and implementation of AspectD, and a deep analysis of the issues in using Oracle’s implementation of the standardized Java Platform Debugger Architecture (JPDA) [10], in particular its Java Debug Interface (JDI) API [11], to unobtrusively connect the original application to various kinds of monitors via aspect-oriented techniques.
- Implementation strategies for pointcuts, using a variety of types of breakpoints (called breakpoint strategies), as well as an analysis of their performance.
- Three case studies evaluating the expressiveness and performance of AspectD, including an implementation of a new front end for the Daikon invariant detector [12].
- Prototype enhancements to the JPDA showing that small changes to its implementation can provide substantial performance improvements.

To preview our results, our experiments demonstrate that a small set of well-defined primitives for setting and removing breakpoints and changing program state suffice to create an expressive aspect-oriented debugging facility. However, the APIs available in the debugger strongly influence the performance of the aspects. While it is feasible to use a programmatic debugger like the JDI to add aspects that run in a remote process, this slows down the evaluation of complex conditional breakpoints, which are evaluated in the debugger process instead of the application process. We discuss the different options available in the current JPDA to identify join points and transfer the information to the debugger’s JVM, and developed three different strategies to use debugger breakpoints to identify join points.

Although the performance of AspectD is acceptable when the number of join points visited in a run is not too large, substantial slowdowns can occur when join points are hit continuously. Some of these performance issues are expected; they are the price to pay for interprocess communication. Beyond this limitation, however, the JDI is not optimized to support a large number of breakpoint events, even though its programmatic debugging support would seem to encourage such use. We analyzed the current JPDA implementation and identified ways to improve its architecture to boost the performance of some pointcuts that do not currently perform well. Our analysis shows that to scale AspectD to a large number of join points for complex pointcuts changes must be enacted in the API and in the implementation of the debug protocol transport plugin. We also demonstrate some simple changes to the JPDA’s implementation that show that these performance problems are surmountable.

The remainder of this paper proceeds as follows. In section 2 we discuss related work. In section 3 we present the architecture of AspectD; in this section we explain how to use a debugger to apply aspects using breakpoints and we discuss the minimum set of functionalities the debugger must provide. Section 4 presents the relevant implementation details of our AspectD and the tradeoffs we made in the name of efficiency. Section 5 presents three case studies that demonstrates the power of the AspectD language. Section 6 presents three different strategies we use to implement pointcuts. Section 7 analyzes the performance of AspectD. Section 8 discusses the requirements a debugger API must satisfy to support the creation of an efficient dynamic weaver. Finally, we present our conclusions and outline future work in section 9.

2. RELATED WORK

Beyond the general milieu of applying AOP to debugging, tracing, and profiling by adding aspects to a program, there are three relevant threads of research. One is providing support for runtime weaving to provide dynamic AOP semantics for application programming. The second is applying aspect-oriented techniques to an actual debugger, which of course requires support for runtime weaving. The third is event-based debugging. We discuss each in turn.
2.1 Support for Dynamic AOP

There are three broad approaches to runtime weaving, each entailing distinct tradeoffs. All of the below are dynamic extensions of AOP, rather than debugging applications.

Static compilation with dynamic class loading. Spring AOP [13], for example, supports a subset of AspectJ functionalities but weaves aspects by using dynamic proxies created at load-time through reflection. The approach provides a familiar language (AspectJ), with the benefits of selective runtime loading. The success of Spring AOP shows that programmers are sometimes willing to sacrifice the additional expressiveness of AspectJ for the ability to weave aspects at runtime.

Using debugger facilities. There are a few tools we know of that use Java debugging facilities to implement dynamic AOP languages. The first version of PROSE used the Java Virtual Machine Debugger Interface (JVMDI) a native interface that enables the creation of plugins for the JVM [14]. Axon also uses the JVMDI to implement dynamic weaving of aspects [15].

One key difference between our work and these solutions is that AspectD uses the JDI. The JDI enables a programmer to write a weaver (actually, aspects) directly in Java, execute the code in a different JVM, and does not require a modification of the JVM running the base code. On the other hand, the JVMDI supports the creation of native plugins that must be inserted into the JVM executing the base code. This means that the aspect language could be potentially faster but at the cost of a more invasive, potentially perturbing modification of the runtime environment.

Another dynamic weaver based on the Java debugger is Wool. Wool uses two different strategies to weave advice: either using breakpoints, similar to AspectD, or runtime bytecode replacement to insert a hook for advice (see the next paragraph). The first strategy, implemented using the JDI, can be used when join points are hit infrequently and the second when join points that are hit often to avoid the overhead of breakpoints and interprocess communication [16].

Bytecode modification. Most dynamic AOP languages for Java we know of use some form of bytecode or code modification. Beyond Wool, later versions of PROSE changed the implementation strategy from using the Java Debugger to modifying the Java bytecode, thus improving performance [8], [9]. JAC uses Javassist to change the bytecode of classes at load time [17], and Steamloom uses the BAT (Bytecode Augmentation Toolkit) to modify the code in the Jikes Virtual Machine. Other approaches such as EAOP introduce hooks for pointcut events directly in the source code by means of a preprocessor [18]. HotWave supports the weaving of aspects both at compile time and at runtime. With the latter, it supports (re)weaving of classes that were previously loaded. For this purpose, HotWeave changes the classes’ bytecode and uses Java’s hotswapping capabilities to replace the original classes [19]. In this respect, HotWave is similar to Wool.

2.2 Aspect-Oriented Support for Debugging

Bugdel is the first tool that we know of that practices AOD [20]. Interestingly, rather than building on a debugger, the authors add debugging features to an aspect-oriented programming language (AspectJ) and then add a GUI front-end that plugs in to Eclipse. Bugdel adds joint points for line numbers, allows advice to access local and private variables, and adds useful information to AspectJ’s thisJoinPoint, such as file name, line number, and the name-value pairs of all local variables and fields. It also provides a breakpoint-like feature by having the application make a call-back to the IDE at pre-defined points. The direct-manipulation GUI reduces the need for the programmer to know the syntax of AspectJ.

Since Bugdel is adding debugging features to AOP, it follows the trend in dynamic AOP of using bytecode modification. Specifically, Bugdel implements weaving using Javassist to modify the program bytecode. This improves performance, but carries the disadvantages of bytecode modification discussed in the Introduction: missing dynamically loaded classes, calls through reflection, and code introduced or modified by the application’s own bytecode modification subsystem.

In general, Bugdel and AspectD represent opposite approaches to managing a set of complex tradeoffs. Bugdel has access to the complete power of AOP (e.g., all of AspectJ’s join points, plus its own), whereas the AspectD approach is generally limited to the wildcards and breakpoints provided by the language’s debugger. AspectD has all the features and benefits of a hardware-supported debugger, including interactive use and non-intrusive monitoring, whereas Bugdel sacrifices these for the full power of AOP.

2.3 Event-Based Debugging

In event-based debugging, callback procedures handle events generated when a running application reaches selected program points. Most debuggers of this kind are equipped with a scripting language to support debugging. Early efforts in this area are Event-Based Behavioral Abstraction (EBBA) [21], Dalet [22], Coca [23], Acid [24], and Dispel [25].

The MzTake family of scriptable debuggers builds on these previous ideas and supports Java and Scheme [26]. MzTake defines a language based on FrTime – a dataflow language with Scheme-based syntax. This language supports representing sequences of events as lists. Programmers can then apply standard functional list operators, such as map and reduce, to manipulate debugger event streams.

Expositor [27] builds on the line of work of MzTake. Expositor improves on the performance of previous work and provides a powerful tool for programmers to understand the flow of their programs. In particular, Expositor offer the ability to look up events in the past.

AspectD is event driven, but employs AOP to concisely specify join points with abstract pointcuts and decouple aspect code from them. Also, our aim is to create a general programming framework for interacting with existing programs using a language’s standard debugger. Ultimately, this includes the making of changes to the state of the running application, a capability that runs counter to event-based debugging.

3. THE ASPECTD ARCHITECTURE

AspectD’s design follows the architecture depicted in Figure 1 to create aspect-oriented debugging programs; aspects are weaved to the application at runtime using a debugger. Following this architecture and using different debuggers, we could create aspect-oriented debuggers for different programming languages. We call the original application that is under debug the Subject Application and the aspect code weaved using the debugger the Debug Application. The Subject code is unmodified and debuggers use special hardware supported instructions to observe and interrupt the Subject’s execution.

Debuggers support the creation of conditional breakpoints, points in the program code where the execution will stop given certain conditions. In AspectD, we write applications that use programmable debuggers to define breakpoints on the Subject.
Once the execution hits a breakpoint, the Subject stops and an event informs the Debugging application that the breakpoint was reached. AspectD defines a simple language to specify pointcuts and advice, pointcuts are translated into breakpoints and applied to the Subject application. When a breakpoint event is generated by the debugger, then the corresponding advice code is run in the Debug Application. The advice code can use the debugger facilities to change the state of the Subject application. The Subject application resumes execution once the advice completes.

The first element on the top left of Figure 1 is the Subject Application. The Subject is any application that can be debugged with the chosen debugger and does not need any special modification to support AspectD.

AspectD has two parts: 1) the AspectD Runtime uses the debugging API to connect to the Subject and perform the debugging; 2) the AspectD compiler receives the pointcut definition and the code implementing the advice; it then generates a Debug Application.

The Debug application interacts with the runtime to define the set of breakpoints that represent the pointcut and receives event messages every time the Subject executes one of the join points (implemented as a breakpoint). The result of the advice execution is usually an analysis of the running system. Advice can use the debugger APIs to change the Subject’s variables values or the application flow.

A programmable debugger that supports AspectD must provide an API for setting up breakpoints. The API must generate events (or callbacks) that report when the Subject reaches a breakpoint. The runtime part of AspectD provides a pluggable architecture that allows replacing the breakpoint monitoring strategy used to obtain the join point events. We make the strategy used to convert pointcut definitions into breakpoints pluggable in order to support multiple implementations. For example, different debugger implementations support a different set of conditional breakpoints. Also, choices of different strategies within a given debugger enables minimizing the overhead introduced by the debugger on the running application. In fact, depending on the expressiveness of the API exposed by the programmable debugger, it is possible to monitor some events without interrupting the execution of the Subject Application. This substantially improves performance. Other kinds of events require the interruption of the VM execution and the execution of code in the AspectD Runtime, leading to slower performance.

We defined a simple language to describe pointcuts and advice for AspectD applications. The language’s primary job is to declaratively connect debugger commands to functions that run on the data captured by those commands. An AspectD program is a sequence of advice definitions. Each such definition has (a) an event type and a regular expression (the two comprising a pointcut), (b) the advice action, which is a method invocation, and (c) a variable that is bound to values captured by the pointcut and passed to the action.

Event types are language-specific so different language implementations can have different event types. Figure 2 presents the BNF of our advice language for Java. It has three types of events. 1) A field.access event defines any reading from or writing to a field in the Subject. 2) A method.enter defines the execution of a method entry point in the Subject. 3) A method.exit defines the end of a method execution. Exits comprise both normal returns and exceptional returns. The regular expression specifies the names of elements in the source code whose events are part of the pointcut. The final pieces of the advice definition are (a) the method to run when a join point corresponding to a given method and regular expression is reached, and optionally (b) an initialization method (start) to be called before the first event is delivered and a finalization method (stop) to be called after the last message. Advice, start, and stop methods are written by the developer of the aspect in the specific language supported by the tool implementation.

Because AspectD uses the debugger to weave aspects it can support unique types of debugging-oriented pointcuts. For example, our pointcuts can be defined on private fields and methods. An experimental plugin even supports pointcuts on specific code line numbers and advice that can access local method variables.

4. IMPLEMENTATION DETAILS
The AspectD implementation is based on the Java Debug Interface (JDI) API. An interesting aspect of this tool is that the method implementing the advice runs on a separate Java Virtual Machine (JVM). The results is an aspect-oriented program that can be dynamically composed into a running application and removed without changing any of the code running in the JVM, minimizing unwanted side effects.

The Java Debugger has two modes of operation: 1) it can run the Subject application using the debugging facilities of Java, or 2) it can connect to a running Java application launched with debugging enabled and listening on a given port for a debugger to connect. This second option gives the tool flexibility for runtime debugging and monitoring of applications. A developer can connect to a running application and analyze it only when needed. After finishing her analysis, she can disconnect the debugger to avoid overhead while the subject application keeps running.

The current version of our tool does not provide an alternative API to change the execution of the Subject. Therefore, applications must use the JDI API in their advice code to modify the Subject’s runtime behavior. This does not limit AspectD expressiveness.
fact, using the debugging facilities of Java it is possible to change the program state and flow. The debugger API supports changing variables and fields values, returning early form a method call with an arbitrary value, and many other ways to modify the state of the Subject JVM.

The regular expressions used in the pointcut language of AspectD is what JDI calls a restricted regular expression, and matches fully-qualified class names. It can either be the exact name of the class to which we want to apply the aspect, or it can use one dot and one * wildcard character to replace the first part or last part of the name. In this case, all classes matching the final part of the class name or the first part of the class name respectively will receive the advice. To simplify the implementation, we directly adopted the JDI’s restricted regular expressions, but our advice translator could be enhanced to take more general regular expressions and translate them into multiple restricted regular expressions.

The implementation of advice, start, and stop in AspectD uses fully qualified names of static methods provided by the application developer. We give developers the option of using the method name (short name) in the advice file and to import the fully qualified class name using a command-line parameter of the compiler. We use this option in the examples presented in this paper. We use the variable name in the definition to identify the parameter of the advice that will receive the event. Future work will allow the pointcut to extract values from the event variable to support advice actions that are oblivious of the event class structure used by AspectD.

The AspectD compiler translates the program into multiple Java classes that mediate the interaction between the AspectD runtime and the classes implementing the advice. The AspectD runtime, the classes generated by the compiler, and the classes implementing advice together form the Debug Application. The compiler generates classes extending either ApplicationFactory or ApplicationObserver interfaces. The compiler creates one factory whose responsibility is to initialize and return instances of the observer classes, and the set of regular expressions identifying the classes that are part of the pointcut. Each advice line in the file originates an Observer class. We use a command-line parameter to tell the AspectD runtime which factory class to load, and consequently which debugger application to run. The AspectD runtime sends all events to the Observer objects; in turn each Observer dispatches the event to the proper advice.

The Pointcut Language

The name in the definition to identify the parameter of the advice that we are interested in field access and defines what classes should be monitored. A single advice action is implemented in a static method called spy. Assuming we want to spy the fields of all classes in the package edu.ucsd.delphi and its sub packages we use the following advice:

```
@access:field.access:edu.ucsd.delphi.*; => spy(@access)
```

When AspectD compiles this advice it generates the two classes in Figure 4. The first one, FieldSpyAdviceClass, is the concrete implementation of the ApplicationObserver interface. As explained in Section 4, it mediates the interaction between the AspectD runtime and the advice action. In this example, the method parse of FieldSpyAdviceClass calls the spy method. However, before calling spy, parse verifies that spy is interested in the event, checking that the regular expression and the type of event specified in the advice match the event received. The AspectD runtime creates one event for every breakpoint notification generated by the Subject VM. One responsibility of the Observer is to ensure that only events of the relevant pointcut reach the action method. The second class that FieldSpy implements is ApplicationFactory. The two methods generated by the compiler are: getClassesToDebug, which returns a list with all regular expressions defined in the advice, and onlineParsers, which returns instances of the Observers defined in the Debug Application. Because AspectD uses the debugger to monitor the field changes, this application can monitor any field, even private ones.

5. Method Call Analysis

A simple analysis we can perform is to identify if one method gets called during the execution of another. For example, assume we have a set of test cases and we want to identify which methods each test calls. The result of our analysis will tell us how many tests...
“touch” a given method, and therefore, what regression tests to run when we change one method. The implementation of this application in AspectD is simple. We request the entry and exit events for our test classes and the entry event for the class that contains the method we are interested in analyzing.

The advice for our application is the following:

```java
@e:method.entry:edu.ucsd.hello.*; => callers(@e)
\startCallAnalysis /endCallAnalysis
@endCallAnalysis
```

The first line (rendered on 2 lines in this manuscript) identifies the methods we are analyzing, while the other two enable and disable the call tracing ensuring the application analyzes only class Test. In this example, we check if any method of the Test class calls any method of any class in package edu.ucsd.hello or in its sub-packages. The result we expect from this analysis is a list of methods called and for each of these methods a list of Test methods that call them. To implement this analysis, we create a set of stack structures (one per thread of execution in the Subject application). Each time a new method of Test starts executing, the tracking advice adds it to the stack of the proper thread. Every time the method returns, the tracking advice removes it. Whenever a method in a class matching the first regular expression is entered, our application invokes the `callers` method. This advice checks the current stack structure created by track. If it finds any methods in the stack, it adds these methods to the set of tests that call the current method. The `startCallAnalysis` method is used to initialize the state when the application starts. After delivering the last event to the `callers` method, AspectD calls `endCallAnalysis`. Our application uses this method to output the result of the analysis.

It is important to notice that the order in which the advice are specified matters. AspectD supports two different strategies for calling advice, online and batch. This application uses the online strategy. With it, the runtime delivers each event as soon as it is available to all advice. The order in which the events are delivered is deterministic and advice are executed in the order in which advice are specified in the file. Changing the order of advice in the file can lead to different results.

If we change the order of the first two lines in our case study, for example, we get the wrong result. In fact, there is overlap between the pointcut used for tracking Test calls and the one used for identifying dependencies. Track will add any method in Test to the stack. This is fair because we are also interested in identifying if methods of Test call other methods of Test (or even themselves recursively). However, by changing the order of the advice, any method in Test will be added to the stack before invoking the advice action that checks dependencies. Thus, every method in Test would be erroneously reported as calling itself.

### 5.3 Java Front End for Daikon

This case study is substantially more complex that the previous two and demonstrates how to create a complex analysis application using AspectD.

The Daikon invariant detector is a tool that finds likely method invariants [10]. Daikon processes data traces created by front ends. Different front ends instrument applications written in different programming languages and create language-independent trace files. Trace files record the entry and exit states for program methods during execution. A list of Daikon front ends is provided online [12]. Not surprisingly, a Daikon front end can be quite complex, and the one for Java, Chicory, is no exception. Despite its complexities, Chicory does not capture complete trace data for methods that generate exceptions, resulting in missing invariants. This is an artifact of its instrumentation being embedded in the application, as opposed to using remote monitoring. Nor does Chicory fully track dynamically loaded classes or code that uses reflection. Finally, the libraries and data structures that Chicory employs sometimes cause the instrumented program to fail.

We developed an alternative Daikon front end for the Java language that solves these problems, yet is considerably simpler than Chicory. Being a prototype, however, our front end omits some of the features of the Chicory, such as support for array data types. Because of space constraints we do not show any code for our frontend here. However, this application is bundled with AspectD distribution.

Our implementation strategy is to specify what class we want to analyze and intercept all the field accesses and method calls that the class’s methods perform during execution. Because of the way Daikon traces are structured, we created two different applications that work together to create a full Daikon trace.

The first application collects information for all method calls and returns, and generates a set of entry and exit declarations. These must appear in the trace file before the relevant data points. Because debug applications instrument methods on the fly and observe effective calls, our front end cannot know a priori what these declarations will look like. The first part of the front end then records all relevant data, and when the Subject terminates the second part generates the proper declarations for all methods in the run.

After creating all the declarations, our debugger application runs a second application that processes again all the application events and generates the proper data points for the Daikon trace. AspectD supports this feature by recording all events of the Subject run. After the subject application exits, the debugger runtime reads all events of the previous Subject run from a temporary file and sends them to the application advice action. We call this a batch advice. An application registers itself with the runtime to receive a batch advice, once the program debugging session terminates all relevant events are played back to each application registered batch advice.

The Daikon front end is more complex not only because we perform the analysis in two phases (i.e., passes), but also because the pointcut we need to specify for the data we are interested in is more complex. In fact, we not only need to intercept calls and returns for methods of the specified classes. We also want to know what methods are called directly by all methods our pointcut identifies. In the next section, we describe one of the breakpoint strategies we defined for the AspectD runtime that optimizes this complex pointcut.

### 6. Breakpoint Strategies

The applications discussed in section 5 use different pointcuts and need different amounts of information on the Subject run in order to work correctly. AspectD uses breakpoints to stop the execution of the Subject at the pointcut and pass the control to the advice. Many possible strategies exist to identify which join point reached at runtime are part of the pointcut. AspectD splits this work between the debugger, which sets breakpoints in the Subject and notifies AspectD runtime when the execution reaches them, and the runtime, which can filter out unwanted breakpoints.
In theory, we could analyze every instruction executed by the Subject application and let each of the Observers, running in the AspectD runtime, decide if the event is relevant. This strategy would set a simple single-instruction-step breakpoint, supported by all debuggers that we know of, but would kill the performance of the Subject Application. The ideal solution would be to have the debugger set only the breakpoints that correspond to the pointcut. In this situation, the Subject application would run at full speed and stop only when the Debug Application needs to run an advice, minimizing the overhead. Unfortunately, it is impossible to implement this solution. The expressiveness of conditional breakpoints in real-world debuggers is limited, and does not allow for the complex pointcuts we can specify in AspectD. An additional problem is that the performance of different types of breakpoints varies substantially even in the same debugger; therefore, we sometimes write plugins with complex strategies even if a simpler set of breakpoints would suffice to achieve better performance.

We devised different strategies for setting breakpoints and encapsulated these strategies in plugins that a programmer selects when developing a Debug Application. While the most general one, which we use for our Daikon Front end for Java, can support all our test cases, the use of simpler breakpoint strategies, which return fewer events and less information for each event, provide a performance boost when our applications do not need the additional information.

The **Field Modification Plugin**. Figure 5 shows the strategy used to implement Field Spy’s pointcut using AspectD. This is the simplest plugin we developed for the runtime module. It adds breakpoints on field modification for all fields of classes that match the class regular expression. These breakpoints are defined when the class is loaded into the Subject JVM. This is the only operation that blocks the execution of Subject. Because classes are typically loaded once the first time they are used, the overhead of this operation is low in comparison to the amount of time spent in the execution of the Subject application. Every time a field with a breakpoint is modified, the Subject JVM notifies the AspectD runtime of the new value, without interrupting the execution. The runtime, running on a different JVM will in turn create an event and via the Observer generated from the pointcut, invoke the proper method.

The **Method Call Return Plugin**. The Method Call Analysis application does not need information about field access. The only events we need for this application are notifications when the Subject enters and exits from the given methods. The simplest debug strategy plugin we can use for this requirement introduces a breakpoint in the first instruction and one for each return instruction of every method identified by our pointcut. Figure 6 depicts this strategy. This strategy is fast because the Java virtual machine of the subject debugger can asynchronously notify the debugger that the breakpoint was hit while it keeps running the Subject application. Moreover, it uses instruction point breakpoints, which as discussed in our analysis later in this section are fast. The figure shows the activity the plugin follows to instrument the Subject JVM with breakpoints, and generate events for advice when certain breakpoints are hit. The plugin blocks the Subject execution when the class is loaded; this usually happens once per application run. In this phase, the plugin defines the breakpoints on the method entry and return points.

The only issue with this approach is that exceptions can break out of methods without hitting the return points. Our strategy also intercepts exceptions and generates a proper return event when some method in the pointcut would not execute its return instruction because the exception is not caught locally. In Figure 6 we depict this process. When the debugger detects an exception, it blocks the execution of the subject and creates a new breakpoint to the point in which the Program catches the exception. Then the debugger restarts the execution and waits for the Subject to reach the position where it catches the exception. When this happens, the debugger suspends the execution again and generates return events for all the methods that did not properly return because of the exception. The method exit event AspectD properly records the fact that the return was due to an exception. After generating the exit messages, the plugin removes the breakpoint at the catch location and the runtime restarts the execution of the Subject.

Finally, it is important to notice that all the call, return, and exception breakpoints carry the ID of the thread that was running when the Subject encountered the breakpoint. This enables AspectD to support the analysis of multithreaded applications.

The **Full Call Trace Plugin**. The breakpoint strategy used by the plugin supporting the Daikon Front end’s initial workflow is similar to the one discussed in the previous example. After the breakpoint on the entry point of a method is reached, however, the strategy becomes much more complex. In the interest of space,
instead of presenting the detailed diagram of the plugin we summarize the key element of this strategy. With this plugin, we want to capture much data describing the event. For example, we want to know the value of the arguments when we enter a method and the return value when we exit. For this reason, we need to stop the execution at each point and add more breakpoints or query the current program stack to gain information. When the subject application enters one of the methods in a class that matches one of the regular expressions, the application debugger sets multiple breakpoints in that method code. All call instructions and return instructions receive a breakpoint. Now the debugger can trace calls to other methods and returns from the current method. The debugger removes these breakpoints once the current method returns. While we could leave the breakpoints, or add them when the class is loaded, the current implementations of the Java virtual machine slow down and crashes when our application defines too many breakpoints concurrently.

This breakpoint strategy plugin retrieves call parameters and return values for each method called by a method from classes matching the regular expression. To this end, because there is no simple way to specify this in the JDI, after each call breakpoint the debugger steps into the method, breaks the execution and queries the subject stack to get the value of all call arguments. Then the debugger instruments all return instructions and waits for the method to return. When the return breakpoint is reached, the debugger enables a new MethodExit breakpoint and removes the other breakpoints on the current method. The MethodExit breakpoint provides the return value of the method to the debugger. The plugin must cope with exceptions in the Subject application. Moreover, the plugin supports reentrant calls. When a method matching the regular expressions is called while executing another, the plugin disables the current breakpoints, saves the current state in a stack structure, and repeats the breakpoint strategy for the new method.

Some of these plugins are very complex and the reader could wonder if there is an easier way to achieve the same pointcuts using the Java debugger. The JDI supports certain breakpoints such as the MethodEntry and MethodExit breakpoints that would substantially simplify the Full Call Trace Plugin. However, these breakpoints perform badly in the current implementation of Java, thus we could not use them. Plugins are part of AspectD and application programmers will use existing ones. Still, the architecture of AspectD enables writing new plugins to support more pointcuts or optimize performance when needed.

The complexity of these plugins is due to the fact that the JDI API is not optimized for aspect weaving. As we discuss in the next section, much of the complexity of these plugins arises from two problems: 1) performance issues with some of the breakpoint types, and 2) missing API functions that would allow the AspectD runtime to be notified of interesting events. An example of this second type of limitation is that it is impossible to be notified when a method exits because of an exception. Our plugin must capture the exceptions and reconstruct the state of the call stack to determine if the method being monitored has exited.

It is also important to notice that, while the programmer can select the most appropriate strategy to fine tune the performance of his application, there is no need to select a strategy. If no strategy is specified the runtime uses the most complete one, making all the event available to the aspect application. It is future work to improve the compiler such that it can automatically select the fastest strategy supported by each application.

7. PERFORMANCE ANALYSIS

The differences between these three breakpoint strategies have a substantial impact on performance. As expected, adding aspects slows down the Subject application. This effect can be minor or substantial depending on the breakpoint strategy used and the application under consideration. The last strategy we discussed can be substantially slower than the first two.

Factors that affect the performance of AspectD are:

1. Breakpoints that force the JVM to interpret part of the byte code instead of using the just-in-time compiled version.
2. Overhead introduced into the JVM by each type of breakpoint.
3. Number of breakpoints generated by a given strategy.
4. Type of protocol used for interprocess communication between the Subject and the Debugger VMs.
5. Delays introduced by a breakpoint blocking the execution while waiting for commands from the debugger.

In our experiments, we readily identified that some types of breakpoints can slow down the execution of the VM independently of the number of times they are hit during a run. For example, in the current implementation of the JVM, when MethodEntry and MethodExit breakpoints are defined, the execution slows down several orders of magnitude even if no breakpoint is ever reached.

Table 1 summarizes measurements we ran on a simple test program using a Java debugger to assess the effects of different types of breakpoints on the current JVM. We used the IntelliJ IDEA 13.1.4 IDE and the default transport protocol (socket transport) to run these experiments. The results are averaged over 100 runs for shorter executions (up to 1 second) and over 10 runs for longer ones. We ran all tests using Oracle JDK 1.8.0_20 64bit on a Windows 8.1 machine. The hardware was an Intel Core i7 Sandy Bridge running at 3.4 GHz (model 2600K) with 8GB of RAM.

The first observation is that the cost of a breakpoint depends heavily of the type of breakpoint we use. For example, line breakpoints have a negligible cost when they are not hit. On the other hand, the use of field breakpoints slows the execution substantially even if the field with the breakpoint is never accessed. In our experiments, the Subject program slowed down more than 80 times just for enabling one breakpoint that monitors field accesses. As a frame of reference, we measured the execution time of our test application with the just-in-time compiler disabled. The result, on the second line of Table 1, shows that while we get a substantial slowdown,
more than 25 times the execution time with the just-in-time compiler enabled, the execution is still more than 3 times faster than the execution with a field breakpoint enabled.

We also focused our analysis on the scalability of each breakpoint hit during the execution. We stress-tested our debugger using field and line breakpoints. In our measurements, the difference between hitting 1 and 100 line breakpoints is 17 milliseconds for breakpoints that do not stop the JVM execution and 31 milliseconds for breakpoints that stop the JVM execution until the debugger process receives the debug information and immediately restarts the execution. In this second case, the additional time is due to the lag induced by the roundtrip between the Subject and the Debugger application. Our results for 100,000 breakpoints show that the difference between field and method breakpoints is really in the JVM logic that implements the breakpoint. In fact, if we look at the execution time with 100,000 hits of these two types of breakpoints, we see very little difference if we discount the slowdown incurred just by enabling the field breakpoint.

These results guided us in creating the three plugins for AspectD. In particular, we tried to depend heavily on line breakpoints, by instrumenting the first instruction and all return instructions of our methods. We still rely on method exit breakpoints to obtain the return value of a method. However, to minimize the effect of having such breakpoints enabled, we create and remove these breakpoints on the fly, enabling them only during the single step of returning from a method call.

Because AspectD executes the advice action in a different VM than the one running the Subject application, we focused our analysis on the performance of the AspectD runtime module and the different breakpoint strategies and not on particular Debug applications. For our tests, we developed an empty application that does not run any real advice code. Still, it provides regular expressions and uses the three plugins to instrument the Subject with breakpoints and generate the events in the Application VM.

Table 2 collects the results of the performance measurements for our plugins. Our Subject is a simple application with a few loops that runs in 38 milliseconds when the debugger is not connected. The first line of the table shows the results when running AspectD with a pointcut that specifies classes not loaded during the execution. The impact of our tool is negligible in this case. Even if the pointcut identifies classes that are never loaded, AspectD generates a few events. They are the result of breakpoints used by AspectD engine to initialize the plugin strategies. If the pointcut includes classes loaded during the execution, both the plugin that monitors fields and the complete one show a substantial slowdown due to the fact that they both enable some breakpoints on fields, which in turn slows down the execution of the JVM as expected from the data of Table 1.

We also ran AspectD with a pointcut that creates breakpoints that get hit a large number of times during the execution. The last 3 lines of Table 2 show measurements for this pointcut. The difference between these 3 lines is in the protocol used in the communication. The first uses shared memory IPC, which the Oracle JVM supports on local debuggers for Windows systems. The second uses a socket to communicate with both VMs on the same physical computer. The last line shows the results of running the debugger on a separate computer. We connected the two computer to the same local network via a WiFi N connection, which introduces substantial lag. As expected, the results using the shared memory transport are better than the ones using the network. The improvement is substantial for the full plugin. The reason is that the full plugin stops the execution of the Subject VM every time it reaches a breakpoint, then it adds or removes breakpoints. The shared memory solution reduces the communication delay, and therefore performs much better. The results on the last line show that, while remote monitoring and debuging is doable, we should only use plugins that do not stop the execution of the VM. Our most complex plugin increased the execution time to more than one hour when tested on a WiFi network, more than 30 to 50 times longer than what it took to run the application locally. In comparison, running the field monitor plugin remotely increased the run time by only 4 times.

We ran our three Debug Applications on real Java programs. In particular, we used two open source projects: Apache Commons Math [28] and Apache Tomcat [29]. Instead of monitoring these applications during a regular execution, we ran them on the testcases that came with these projects. The benefit of doing this, is to have repeatable results. Because these projects use the Maven project management framework [30], we can easily compile them and run all the unit tests. Moreover, Maven has a simple command-line parameter that enables the debugger for all test cases it runs. The results we obtained are consistent with the ones presented in Table 2 when using the empty app.

An additional cause of slowdown are online debug applications using plugins that stop the execution of Subject. Online applications receive and process messages as soon as they are available. In our implementation, we chose to let online debug applications perform the processing of events in the same thread that receives them. This makes sense because online applications change the state of the Subject. For this to be possible, the Subject

<table>
<thead>
<tr>
<th>Pointcut regular expression matches</th>
<th>Field Plugin</th>
<th>Method Plugin</th>
<th>Full Plugin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duration ms</td>
<td>Duration ms</td>
<td>Duration ms</td>
<td>Duration ms</td>
</tr>
<tr>
<td>Events num</td>
<td>Events num</td>
<td>Events num</td>
<td>Events num</td>
</tr>
<tr>
<td>A class not loaded</td>
<td>5</td>
<td>38</td>
<td>17</td>
</tr>
<tr>
<td>A class loaded but not used</td>
<td>6</td>
<td>1,321</td>
<td>26</td>
</tr>
<tr>
<td>A class whose method/field is called/modified 100 times</td>
<td>106</td>
<td>1,342</td>
<td>218</td>
</tr>
<tr>
<td>A class whose method/field is called/modified 100K times</td>
<td>100,006</td>
<td>4,077</td>
<td>200,021</td>
</tr>
<tr>
<td>A class whose method/field is called/modified 100K times Using a localhost socket connection</td>
<td>100,006</td>
<td>4,514</td>
<td>200,021</td>
</tr>
<tr>
<td>A class whose method/field is called/modified 100K times Using a remote debugger on local wifi network</td>
<td>100,006</td>
<td>18,798</td>
<td>200,021</td>
</tr>
</tbody>
</table>
must wait for the debugger to decide what to change. Of course, if we have a massive amount of messages and the debug application tries to visualize them as they arrive, it will stop the Subject application for a long time. A simple workaround is to use plugins that do not stop the execution or batch advice.

Finally, we designed an experiment to verify our thesis that many performance issues can be addressed by improving the JDI API and its implementation. To this end, we developed a prototypical plugin for AspectD that evaluates conditional breakpoints. The idea was to try a typical debug activity, logging a method call only when the method is called with some given parameter. The pointcut for this activity selects methods to log and the advice prints the call information to the standard output of the debug application. The key difference from within the standard call aspect is that the breakpoint for the method call checks that an integer parameter used in the call is equal to a specified value. Using just the JDI API and not changing the bytecode of the Java application, our plugin must receive notification for all method calls in the debug application process. Then it uses the JDI to query the call parameter and compares it to the specified value. We developed a simple test application that calls a given method 100,000 times. Only a fraction of the calls have the correct call parameter and should must be logged. The execution time of the application we were debugging went from an average of 158ms to 35,997ms when running with AspectD.

To evaluate what performance improvement can be expected by changing the API we modified the current OpenJDK Java VM (version 1.8 u40) to support the evaluation of the condition directly in the VM being debugged. We developed a simple extension that supports just conditions on int values in call parameters. While this is enough to demonstrate the performance implications, more work is needed to support evaluating generic conditional breakpoints locally. The results are very encouraging, the execution time went from an average of 35,977ms to an average of 7,271ms. This shows that there is potential for substantial performance improvements by extending the JDI API. It is also interesting to note that the modification to the VM to support conditional breakpoints was just a quick prototype; optimizing the implementation could lead to even greater speedups.

8. DEBUGGER API REQUIREMENTS

Two important application types do not scale well in the current implementation: 1) applications that perform substantial tracing of the Subject code, and 2) applications that employ complex pointcuts, such as our Daikon Front end, which requires many roundtrips between the Subject and the Debugger JVMs. This is due to JDI API limitations.

The problem with the first type of application is expected. The Java VM is not optimized for sending a massive number of breakpoint events to the debugger. Java provides a second technology, the Java VM Tool Interface (JVMTI), aimed at creating tracing and profiling tools. The debugger itself builds on top of JVMTI. We used it to implement our conditional breakpoint extension and demonstrated that it is possible to improve conditional breakpoint performance by writing JVM extensions.

The second problem is more surprising. The JVM supports a standard protocol to create external debuggers. However, the JDI API is too limited for debugging complex applications. A simple conditional breakpoint cannot be evaluated on the Subject VM. We need to create a generic blocking breakpoint and every time the method executes the VM stops and sends a notification to the debugger. The debugger has to pull the value of the call parameter, check if it matches the constraint in the debugger VM, and if so notify the user; otherwise the Subject resumes execution. In complex applications, breakpoints of this type are the norm. Sometimes the breakpoint is hit thousands or even millions of times before the expected condition is true. As demonstrated in the previous section, using the basic debugging capabilities of the JDI, we cannot handle conditional breakpoints efficiently. This leads to the definition of the following three new debugger requirements.

Requirement 1. The communication protocol should support batch transfer of non-blocking event and out-of-band immediate delivery of blocking events. The root cause of the performance issues we observed is that the current transport plugins are optimized to send a single breakpoint event (or a set of concurrent events) to the debugger. Caching of non-blocking events, and sending them in larger sets would reduce the overhead of interprocess communication. Moreover, non-blocking events should be sent to AspectD from a dedicated thread, letting the code that generated the breakpoint continue executing as soon as the breakpoint notification data is stored in memory.

Requirement 2. The debugger should support defining breakpoints with conditional predicates over the current state of the application and evaluates them locally on the Subject Application. The JDI API’s ability to change the byte code of a method provides an option to inject code in the running program and accelerate this type of scenario. However, this would change the Subject program, defeating one of the purposes of AspectD. A better option is extending the debugger API and the JVM to enable conditional breakpoints being computed on the Subject JVM. This approach does not mix the debugger state and the application state, minimizing the risk of corrupting the application state.

Requirement 3. The breakpoint definition API should support defining what state information must be sent with the notification. The current JDI API forces the debugger to suspend the execution of the Subject while retrieving relevant state information. The ability to specify what information to collect and send when a breakpoint is hit could substantially decrease the number of blocking breakpoints, increasing the application performance.

9. CONCLUSION

We have presented an architecture, a language, and a tool that enable aspect-oriented debugging, without changing the base code and without changing the state of the application being observed. This avoids compatibility problems between the aspects and complex applications and libraries. The application and advice are completely isolated from each other and can run on a separate physical machine, avoiding any unwanted interaction. We evaluated the performance of AspectD and identified technical limitations in the debugger we used. We explained why some types of pointcuts are slow in the current tool and identified three new requirements for debugging APIs. As demonstrated through prototyping, if these requirements are implemented by a programmable debugger, they would make AspectD, and any dynamic aspect weaving tool based on such API scale better. As future work we will complete the changes to the JDI API to support the new requirements. We will implement these extensions leveraging JVMTI and existing work in optimizing dynamic weaving in the Java Virtual Machine.

10. ACKNOWLEDGMENTS

We thank Sorin Lerner for his feedback on the early work that led to this paper. This work was supported in part by National Science Foundation grants SHB-1237174 and CNS-1446912.
11. REFERENCES


