A Software Architecture for Mixed-Language Debuggers

Byeongcheol Lee
UT Austin
bclee@cs.utexas.edu

Martin Hirzel
IBM Hawthorne
hirzel@us.ibm.com

Robert Grimm
New York University
rgrimm@cs.nyu.edu

Kathryn S. McKinley
UT Austin
mckinley@cs.utexas.edu

Abstract

Source-level debuggers improve productivity by helping developers control and inspect program execution. Although many programs use multiple languages, building mixed-language debuggers remains a challenge and debugging mixed-language programs is at best painful. This paper introduces Lamp*, a software architecture for building mixed-language debuggers. Lamp defines a debugger agent and a uniform interface to single-language component debuggers with standard features. The Lamp agent orchestrates execution and single-language component debuggers. To demonstrate the architecture, we implement* Blink*, a debugger for Java and C programs. Blink is portable: it supports multiple operating systems (Windows and Linux), multiple JVMs (Sun and IBM), and multiple C compilers (GNU and Microsoft). Blink is scalable: its new implementation effort scales proportionally to its new features, adding 7 thousand lines of code to 80 thousand for `jdb` and a million for `gdb`. Blink is efficient: debugging is as fast as with `jdb`. These results demonstrate that Lamp is a practical architecture for developing mixed-language debuggers that can greatly ease developing correct mixed-language software.*

1 Introduction

Programmers spend a lot of their time debugging. Debuggers let users examine state and provide fine-grained control over an executing application [9]. Although many applications are written in multiple languages, debugger support for them remains limited. For example, Java programs make many calls to and from native C code, as shown in Table 1. Native code gives Java access to platform specific functionality, legacy libraries, and low-level efficient implementations [6]. However debugging them is currently painful [12, 13]. For instance, if you use just `jdb`, you can not set a breakpoint in your C code. If you manually attach both `jdb` and `gdb`, then you must have the foresight to set all C breakpoints when stopped in C, and vice versa for Java. Neither debugger can inspect the state of other's runtime systems. Worse yet, if stopped in C, `gdb` cannot inspect the Java stack to tell you where you came from.

SPECjvm98 benchmark	Java $\rightarrow$ C	C $\rightarrow$ Java
compress	159	873
db	317	3,939
jack	83	383
jess	300	1,774
mtrt	25,853	53,165
javac	1,828	2,808
mpegaudio	17	144

DaCapo benchmark	Java $\rightarrow$ C	C $\rightarrow$ Java
antlr	19,378	37,782
bloat	228,731	456,564
eclipse	280,959	401,213
fop	11,465	22,566
hsqldb	49	549
jython	361	1,638
luindex	46,537	81,087
lusearch	160,768	1,342
pmd	913	10,420
xalan	25,528	45,961

Table 1. Dynamic language transitions with JNI, using Jikes RVM and the GNU classpath libraries. Sun's and IBM's JVM and libraries yield similar results.

To develop correct mixed-language applications, programmers need a debugger that understands and controls the entire application, supports specification of breakpoints regardless of the current breakpoint language, provides fine-grained execution control when crossing between language barriers, and inspects the state of both languages at a single breakpoint. Building a mixed language debugger is difficult because languages may have different execution modes. For example, Java runs in a managed runtime with just-in-time compilation and garbage collection, while C is generally unmanaged and compiled ahead of time. Prior mixed-language debuggers such as Sun Studio dbx [11], the .NET debugger [10], and XDI [8] extend a single execution-mode debugger.

This paper presents Lamp, a new software architecture for composing debuggers to provide implementation scalability and portability. Lamp defines (i) a standard interface to component single-language debuggers and (ii) a debugger agent to act on its behalf in the debuggee process. Lamp requires a well-defined FFI (foreign function interface), such as JNI (the Java native interface), to be able to interpose in language transitions. Interposition, together with debugger context switching, are the basic functionality on which Lamp builds to provide mixed-mode debugging.

As proof of concept, we implement Blink, a debugger composed of Java and C debuggers. Blink's implementation scales with its new functionality. It adds only 7,000 lines of code to the single-language debuggers (e.g., 80K for `jdb` and 1,000K for `gdb`). Blink supports multiple operating systems (Windows and Linux) and composes multiple debuggers. We show that Blink is portable by demonstrating it with Sun and IBM JVMs, and `gdb` and Microsoft `cdb` C debuggers. As far as we know, Blink supports more platforms than any other mixed-mode debugger. We show that Blink is functional on all supported platforms with a range of feature tests, such as stepping across language boundaries, context and data examination, and mixed-language expression evaluation. We measure Blink's performance and show it has negligible overhead. Blink is more portable than other approaches because it can use platform-specific debuggers, rather than extending a single debugger. These demonstrations show that composition is a practical approach for designing debuggers, which can in turn ease development of correct mixed-language software.

2 Related Work

The closest tools to Blink are Sun Studio dbx and the .NET debugger, which are proprietary commercial *mixed-mode* debuggers that coordinate managed (Java or C#) and unmanaged (C/C++) code in the same system. Unfortunately, neither has published their approach; we summarize based on their user manuals and blogs [10, 11]. Both debuggers are restricted to their respective vendor's operating systems and language implementations. Blink is more portable, running on diverse operating systems (Windows and Linux) and language implementations (Sun and IBM Java VMs, and GNU and Microsoft C compilers). We believe both Sun Studio dbx and the .NET debugger started out as unmanaged debuggers that were then extended with a managed mode, while our Lamp architecture composes managed and unmanaged debuggers without modifying them, and Blink proves the concept.

The XDI research debugger augments the JVM's debug interface to accommodate native debugging of mixed Java and C programs [8]. It is less portable than Blink because it requires JVM modifications not available in standard JVMs. Other efforts rely on orchestrating separate debuggers, but do not provide full support for debugging mixed languages. White shows how to manually orchestrate Java and C debuggers [12, 13], and Chauvin partially automates this process [3]. However, both of these approaches limit the user to a single debugger context, e.g., the user cannot single-step across language transitions or examine state in one language when at a breakpoint in another. Our approach provides full-featured mixed-language debugging, e.g., stepping, state examination, and setting breakpoints, regardless of the current language context. For example, Lamp supports evaluation of mixed-language expressions, whereas none of the prior commercial or research systems can respond to mixed-language queries.

The GNU compiler for Java, `gcj`, indirectly supports mixed-language debugging by precompiling both Java and C code into native executables [2]. But most Java programs are developed on JVMs rather than `gcj`, and this approach is less portable than Blink.

Reentrancy. The Lamp architecture uses an agent, which executes code on behalf of the debugger in the user's process. Like other agent-based debuggers, this architecture is vulnerable to reentrancy problems [9]. For example, the user may set a breakpoint in a C library that the application and the JVM share. The user expects the breakpoint to be reached through a normal JNI call, but this breakpoint may also be reached by code implementing JVM services. For example, IBM's J9 shares some C libraries with applications. It is unsafe to evaluate user expressions during JVM services, since expressions may call back to Java where the JVM may not expect Java code. For example, `malloc()` may be triggered by object allocation and the object will be in an inconsistent state. To detect this case, the agent can keep state about language transitions to ensure that the program reaches each breakpoint through the foreign function interface and only break in this case. We leave checking for this condition to future work. Other agent-based mixed-mode debuggers face the same difficulty [3, 8, 13], and it is a hotly debated topic [10].

3 Lamp Software Architecture

This section describes debugging requirements, mixed-language debugging challenges, and our novel software architecture for composing debuggers.

Standard interactive debugger features. The goal of debugging is to find and correct a defect (erroneous code) that causes an infection (erroneous data) that spreads and leads to a failure (erroneous output) [14]. Rosenberg identifies the following essential features for tool support in this quest [9]:

Execution control: Startup/teardown, breakpoints, and single stepping, e.g., `run`, `break`, `step`, `continue`, `exit`.

Context management: Source code and call stack inspection, e.g., `list`, `backtrace`.

Data inspection: Variable viewing and expression evaluation, e.g., `print`, `eval`.

```
PingPong.java
 1. class PingPong {
 2.   static { System.loadLibrary("PingPong"); }
 3.   public static void main(String[] args) {
 4.     jPing(3);
 5.   }
 6.   static int jPing(int i) {
 7.     if (i > 0)
 8.       cPong(i - 1);
 9.     return i;
10.   }
11.   static native int cPong(int i);
12. }
PingPong.c
13. #include <jni.h>
14. JNIEXPORT jint JNICALL Java_PingPong_cPong(
15.   JNIEnv* env, jclass cls, jint i
16. ) {
17.   if (i > 0) {
18.     jmethodID mid = (*env)->GetStaticMethodID(
          env, cls, "jPing", "(I)I");
19.     (*env)->CallStaticIntMethod(
          env, cls, mid, i - 1);
20.   }
21.   return i;
22. }
```

Figure 1. JNI mutual recursion example.

Mixed-language debugging challenges. Figure 1 shows an example of a mixed-language program. It mixes Java and C using JNI, the Java native interface [6]. Java calls C (Line 8) by calling an empty method (Line 11) implemented by a C function (Line 14). C calls Java (Line 19) by using reflection on a `JNIEnv*` pointer. Debugging mixed-language programs is painful because the single-language debuggers do not provide standard features across different languages. For instance, `jdb` can not set a C breakpoint, and `gdb` can not set a Java breakpoint. To move the program execution from a Java breakpoint to a C breakpoint, the user has to rerun the debugging session, which is at best tedious if the program is slow and at worst almost useless if the program is non-deterministic.

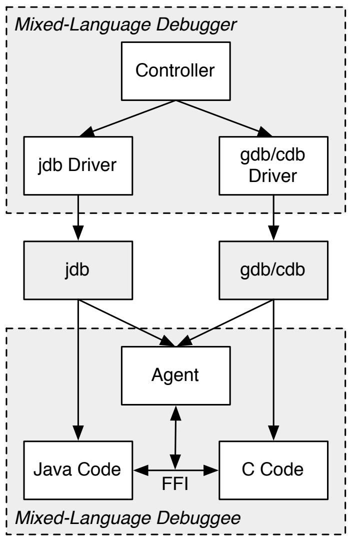

Figure 2. Lamp software architecture.

Lamp architecture and requirements. Lamp composes single-language debuggers and introduces an agent to coordinate them. Figure 2 illustrates this structure. The single-language debuggers support standard features such as breakpoints and expression evaluation for their language, and Lamp builds on those to provide mixed-language debugging. We specify a standard interface to each feature that the component debuggers should provide.

In software tools composition, *scalability* means that the amount of code needed for an extension is proportional to the functionality added [7]. Lamp is scalable because it reuses existing functionality from component debuggers and requires new code only for new functionality. This scalability pays off as *portability*: if the component debuggers are portable, then the composed debugger only requires moderate additional porting effort at language transitions. Likewise, Lamp's scalability pays off as *accuracy* (debugging true to source semantics despite of compiler optimizations [9, 15]): again, the difficult accuracy challenges are taken care of by the component debuggers.

Lamp only relies on component debuggers to control their own language; they do not need to understand any of the other languages' semantics or runtime environment. To manage this isolation, the Lamp agent requires that the interface between the languages is well-defined enough to interpose on language transitions. The agent manages the component debuggers and execution. It must keep track of the current language context and the mixed-language calling context. To coordinate execution, context management, and data inspection at run time, the agent must provide:

Language transition interposition: It must intercept transitions between languages in the program to perform actions on behalf of the debugger.

Debugger context switching: It must transfer control between the single-language debuggers, and between them and the program.

Table 2 correlates the agent requirements with the debugging features they support.

Agent requirement	Interactive debugger feature		
	Context management	Execution control	Data inspection
Language transition detection	X	X	
Debugger context switching	X	X	X

Table 2. Agent requirements vs. features.

Language transition interposition. In order to maintain a fully accurate stack at every breakpoint and to remain in control when single stepping reaches a language boundary, Lamp interposes on all language transitions, which requires a well defined foreign function interface (FFI). The concrete interposition implementation depends on the particular FFI, and Section 4 describes our implementation for the Java Native Interface (JNI).

Debugger context switching. A key challenge for mixed-language debugging is to control execution or examine data in one language while stopped in another. Assuming Java and C, the possible control states are: (i) in Java code, (ii) in the Java debugger, (iii) in C code, and (iv) in the C debugger. If the Java code hits a break point, control goes to the Java debugger. Without an agent, there is no way for the user to, for example, set a breakpoint in C code because the C debugger is dormant and cannot accept requests.

Debugger context switching transfers control between the controller, the component debuggers, and the application, in order to execute user commands that can not be satisfied in the active language. The example in Figure 3 illustrates the difficulty and our solution. Each vertical line in Figure 3 is a process, with the currently active process marked by a box overlaying the line. Horizontal arrows show control transfers between processes. From top to bottom, the application starts out executing Java code and hits a Java breakpoint, which activates `jdb`. Now, suppose the user requests a `gdb` debug action. Without giving control to `gdb`, `gdb` cannot accept any user input. Blink initiates a debugger context switch to `gdb` by using the `jdb` expression evaluation feature to call the debugger

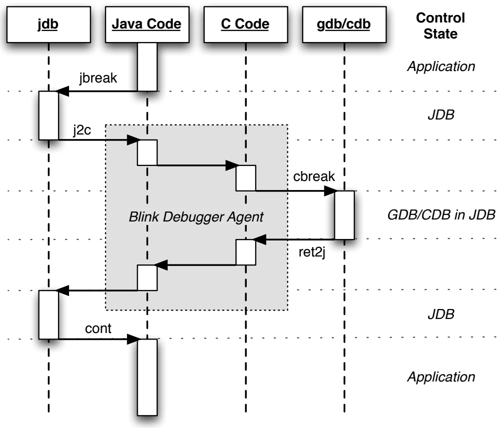

Figure 3. Debugger context switching example, using `j2c` helper function to switch from `jdb` to `gdb/cdb`. Blink also has a `c2j` helper function for switching in the other direction.

agent method `j2c`. The method `j2c` is a Java method that uses JNI to call C, and has a breakpoint in the C part of the code. When execution hits the C breakpoint, `gdb` is activated, and can perform the debug action requested by the user. When complete, `gdb`'s `continue` returns from the C code and Java method, at which point `jdb` wakes up again, since the expression evaluation has completed. The user can either request additional debugging actions in Java or C, or resume normal application execution with `continue`.

Mixed-language expression evaluation. Some debuggers support a REPL (read-eval-print loop), which reads a language expression from the user, evaluates it with respect to the current program state, and prints the result. Lamp generalizes this feature by allowing a mixed-language expression, which nests subexpressions from multiple languages with a language toggle operator borrowed from Jeannie [5]. We are not aware of prior mixed-language debuggers supporting this feature. Section 5 describes Lamp's mixed-language expression evaluation in detail.

4 Blink Mixed-mode Java/C Debugger

Blink is a portable mixed-mode Java/C debugger and an instance of the Lamp architecture. This section presents our experience with implementing Blink.

4.1 Blink Debugger Agent

The Blink debugger agent is a dynamically linked library that includes both Java code and native code executing in the same JVM that runs the user's application. The JVM loads and initializes the Blink agent through the JVMTI [1]. Blink triggers debugger agent actions through the expression evaluation features of the component debuggers: as far as the component debuggers are concerned, these are simply method calls in the application process. The rest of this section describes how the Blink debugger agent satisfies the two Lamp requirements.

Debugger context switching. Blink supports switching the context between its component debuggers as shown in Figure 3. The helper functions `j2c` and `c2j` are part of the Blink debugger agent, and have hardcoded internal breakpoints. The internal breakpoints force the application to surrender control to the respective debugger.

Language transition interposition. The Blink agent must interpose on language transitions to report full mixed-language stack traces and to remain in control when single-stepping between languages. Figure 4 shows the four possible transitions between Java and C.

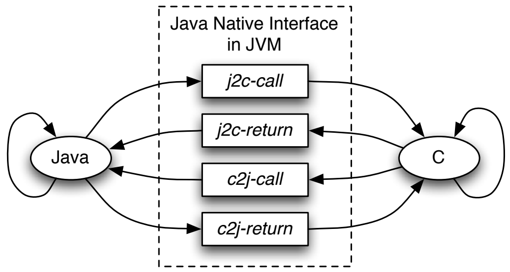

Figure 4. Transitions between Java and C.

j2c-call: Line 8 in Figure 1 is an example of a call from Java to C. It looks just like an ordinary method call, and in fact, with virtual methods, the same call in the source code may sometimes call native methods and sometimes Java methods. But as it turns out, Java virtual machines dispatch all native calls through a central location in the JVM code. The Blink agent discovers this location during start-up by reading a return address from any Java native method body. For instance, the return address at Line 16 in Figure 1 points to the JVM's internal transition from Java to C. To interpose on all j2c-calls, the Blink agent sets a breakpoint at this JVM-internal location.

j2c-return: A j2c-return is a return from a C function to a Java method. It looks just like an ordinary function return, and in fact, the same C function can be called both from Java and C. Blink interposes on this transition by setting a breakpoint in the most recent Java frame on the stack, which it discovers using the JVM's facility of walking all Java stack frames even when they are interspersed with C frames.

c2j-call: All calls from C to Java go through a JNI interface function, such as `CallStaticIntMethod` in Figure 1 Line 19. To interpose on c2j-calls, Blink instruments these interface functions. All interface functions reside in a struct of function pointers pointed to by variable `JNIEnv* env` in Line 15 of Figure 1. During initialization, Blink replaces the function pointers by pointers to wrappers. For example, the wrapper for `CallStaticIntMethod` is:

```
int wrapperCallStaticIntMethod(args) {
  before_c2j_call();
  originalCallStaticIntMethod(args);
  after_j2c_return();
}
```

c2j-return: The same wrappers that interpose c2j-calls also interposes c2j-returns, as shown above.

4.2 Context Management

One basic debugger principle from Rosenberg's book is "context is the torch in the dark cave" [9]. Users, unable to follow all the billions of instructions executed by the program, feel like they are being taken blindfolded into a dark cave in search of a bug. When the program hits a breakpoint, the debugger must provide context.

Source line number information. The most important question in debugging is "where am I", and debuggers answer it with a line number. The Java compilers provide line number information to `jdb`, and the C compiler provides line number information to `gdb` or `cdb`. Blink does not need to add any additional functionality to handle the mixed-language case.

Calling context backtrace. While "where am I" is the most important question, "how did I get here" is a close second. Debuggers answer this question with a calling context backtrace, which shows the stack of function calls leading up to the current location. The JNI code in Figure 1 is an example of mixed-language calls that produce a mixed-language stack. In the beginning, the `main` method at Line 4 calls the `jPing` method with argument 3, yielding the following stack:

$$\texttt{main:4} \longrightarrow \texttt{jPing(3):7}$$

Since $\texttt{i} = 3 > 0$, control reaches Line 8, where the Java method `jPing` calls a native method `cPong` defined in C code as function `Java_PingPong_cPong`:

$$\texttt{main:4} \longrightarrow \texttt{jPing(3):8} \longrightarrow \texttt{cPong(2):17}$$

The C function `cPong` calls back up into Java method `jPing` by first obtaining its method ID in

Line 18, then using the method ID in the call to `CallStaticIntMethod` in Line 19:

$$\texttt{main:4} \rightarrow \texttt{jPing(3):8} \rightarrow \texttt{cPong(2):19} \rightarrow \texttt{jPing(1):7}$$

Finally, after one more call from `jPing` to `cPong`, the mixed-language mutual recursion comes to an end, because it reaches the base case where $\texttt{i} = 0$:

$$\texttt{main:4} \rightarrow \texttt{jPing(3):8} \rightarrow \texttt{cPong(2):19}$$
$$\rightarrow \texttt{jPing(1):8} \rightarrow \texttt{cPong(0):17}$$

At this point, the stack contains multiple and alternating frames from each language. Unfortunately, the single-language debuggers only know about a part of the stack each, because each language implementation uses its own calling conventions. The Java debugger shows all Java fragments, with gaps for C parts of the stack:

$$\texttt{main:4} \rightarrow \texttt{jPing(3):8} \rightarrow \textit{?(C)} \rightarrow \texttt{jPing(1):8} \rightarrow \textit{?(C)}$$

The C debugger has even less information. It only shows the bottom-most C fragment, and knows nothing about preceding Java or C frames:

$$\textit{?(Java/C)} \rightarrow \texttt{cPong(0):17}$$

To recover the missing C stack frames, Blink needs to seed partial C stack walks at the end of each C fragment of the stack, in other words, at calls from C to Java. Using language transition interposition from Section 4.1, Blink instruments j2c-calls to push the current frame pointer and return address on a shadow stack, and instruments j2c-returns to pop the shadow stack. Later, when the user requests a stack trace, Blink obtains partial stack walks from the C debugger starting from these frame pointers and return addresses. Discovering the return address and frame pointer depends on the calling conventions, and is thus nonportable. This amounts to a couple of assembly instructions, which we implemented and tested separately for each platform. For the running example in Figure 1, the partial C stack walks yield:

$$\textit{?(Java)} \rightarrow \texttt{cPong(2):19} \rightarrow \textit{?(Java)} \rightarrow \texttt{cPong(0):17}$$

Interleaving the above with the Java stack yields:

$$\texttt{main:4} \rightarrow \texttt{jPing(3):8} \rightarrow \texttt{cPong(2):19}$$
$$\rightarrow \texttt{jPing(1):8} \rightarrow \texttt{cPong(0):17}$$

Blink thus recovers the full stack and reports it to the user as needed. These implementation details will vary for other languages and their debuggers.

4.3 Execution Control

If context is the torch in the dark cave, then execution control is the means by which the user can get from point A to B in the cave when tracking down a bug. The debugger controls execution by starting up, tearing down, setting breakpoints, and stepping through program statements in the application.

Start-up and tear-down. During start-up, the Blink controller starts the program in the JVM and attaches both `jdb` and either `gdb` or `cdb`. It does this largely by automating White's instructions [12], but in addition, it loads and initializes the Blink debugger agent. To load the agent, Blink uses JVMTI and the `-agentlib` JVM command line argument. To initialize the agent, Blink issues internal commands, such as for identifying the breakpoint location for j2c-call transition interposition. After all processes are set up and connected, but before the user program commences, Blink puts the user in charge with a command prompt. When the program terminates at the end of a debugging session, Blink tears down `jdb` and `gdb/cdb` and then exits.

Breakpoints. Breakpoints are the answer to the question "how do I get to a point in program execution." Users set breakpoints to inspect program state at points they suspect may be erroneous. The debugger's job is to detect when the breakpoint is reached and then transfer control to the user. One of the key challenges for a mixed-language debugger is to set a breakpoint when the location is in a language that is not currently active. This functionality requires the debugger to transfer control to the other language debugger, set the breakpoint, and return control to the current language debugger. Blink takes the breakpoint request from the user, and checks if the request is for Java or C. If the current language does not match the breakpoint language, Blink switches the debugging context to the target language. The breakpoint request is redirected to the Java or C component debugger.

Single stepping. Once the application reaches a breakpoint, the question is "what happens next". Users want to single step though the program, examining control flow and data values to find errors. Single stepping advances the application execution to the next dynamic source line. If the next line is a method call, debuggers offer *step into*, or simply `step`, which steps to the first line in the callee, and *step over*, or `next`, which treats the entire call as one step, executing the application until it reaches the return from the method. The challenge for mixed-language single-stepping is that while jdb can step through Java and gdb or cdb can step through C, they lose control when stepping into a call to the other language or returning to a caller from the other language.

Blink avoids this problem as follows: it sets internal breakpoints at language transitions, so if the current component debugger loses control in a single-step, the other component debugger immediately gains control, and Blink remains in charge. Blink only enables the breakpoints on transitions from the current language to the other language. Furthermore, when the user requests step-over as opposed to step-into, Blink only enables return breakpoints as opposed to both call and return breakpoints. Note that Blink does not make any attempts to decode the current instruction, but rather aggressively sets needed breakpoints just in case the single-step causes a language transition. This approach greatly decreases debugger development effort, since accurate Java single-stepping requires interpreting the semantics of all byte codes, and accurate C single-stepping requires platform-dependent disassembly.

Once Blink enables the necessary internal breakpoints, it implements single-stepping by issuing the corresponding command to `jdb` or `gdb`. There are three possible outcomes:

- The component debugger's single-step remains in the same language. Blink performs no further actions.
- The component debugger's single-step causes a language transition, which an internal breakpoint intercepts. Blink steps from the internal breakpoint to the next line.
- An exceptional condition, such as a segmention fault, occurs. Blink abandons single stepping.

In any case, Blink declares the single-step to be complete and disables all internal breakpoints, as usual for breakpoint algorithms [9].

4.4 Data Inspection

Once the user brought the execution to an interesting point, the main question becomes "is the current state correct or infected." This question is hard to answer automatically, so data inspection answers the simpler question "what is the current state." Blink delegates the inspection of application variables, including locals, parameters, statics, and fields, to the component debugger for the current language, which provides the most local origin for a variable. If, however, the first component debugger does not recognize

```
CompoundData.java
 1. import java.util.Vector;
 2. public class CompoundData {
...
15.   public static native void parse(
        int size, double[] doubles, Vector strings);
16. }
```

```
CompoundData.c
...
20. JNIEXPORT void JNICALL Java_CompoundData_parse (
      JNIEnv *env, jclass cls, jint size,
      jdoubleArray doubles, jobject strings
    ) {
...
```

Figure 5. JNI compound data inspection example.

the variable, Blink tries the other component debugger. Blink also provides more advanced data inspection features described in Section 5.

5 Mixed-language Expressions

The more powerful a debugger's data inspection features, the easier it is for the user to determine whether they are on the right track to a bug. For example, `gdb` provides expression evaluation with a read-eval-print loop (REPL). An interactive interpreter evaluates arbitrary source language expressions based on the current application state. While implementing a language interpreter requires a significant engineering effort, expression evaluation makes it easier to see at a glance whether the current state is infected, especially if the evaluator supports function calls and side effects. Besides debugging, expression evaluation is also useful for testing, program understanding, and rapid prototyping, as users of languages with REPLs will readily attest. Finally, debuggers that offer expression evaluation are easier to compose; for example, `jdb`, `gdb`, and `cdb` are composable into Blink because their expression evaluation permits debugger context switching.

Lamp advances the state of the art of expression evaluation by accepting mixed-language expressions, which nest subexpressions from multiple languages using a language toggle operator. Blink implements mixed-language expressions written in Jeannie [5], which toggles between Java and C using a backtick (`). For example, in Figure 5 Line 20, the current language is C, and variable `strings` is an opaque reference to a Java object. Single-language expression evaluation could only print its address, which is not helpful for debugging. But the mixed-language expression (`strings).size() toggles to the Java language and then invokes the Java method size, returning the length of the Java vector, which is much more meaningful for the user. Mixed-language expression evaluation makes data inspection more convenient.

Lamp requires two basic features in the debugger agent to support expression evaluation:

Convenience Variables: store the results of an expression evaluation in temporary variables.

Mixed-language data transfer: translates and transfers data between the different languages.

5.1 Convenience variables

Application variables are named locations, in which application code stores data during execution. Convenience variables are additional named locations provided by the debugger, which store data for later use in a debugger session. Convenience variables behave like variables in many scripting languages: they are implicitly created upon first use, and have global scope and dynamic types. Besides user-defined convenience variables, debuggers can also have internal convenience variables, for example, to hold intermediate results during expression evaluation. In the mixed-language case, the debugger must remember not only the values of convenience variables, but also their languages. Since gdb already provides convenience variables (written "$var"), Blink just reuses them for storing C values. Unfortunately, both jdb and cdb lack this feature. Blink makes up for this deficiency by implementing its own convenience variables in the debugger agent, using a table that maps from names to values and languages. The table is polymorphic to support dynamic typing.

5.2 Mixed-language data transfer

The Blink agent transfers data from a source language to a target language by first storing it in an array in the source language. It then uses a helper JNI function to read from the array and returns the value in the target language. One complication is that the array and the retrieval function must have the correct type, since the semantics of a value depend on its type and language. For example, an opaque JNI reference in C needs to be converted to a pointer in Java; a struct or union in C, on the other hand, does not have a direct correspondence in Java. Mixed-language data transfer is the only case where Blink must discover enough type information to treat the value appropriately. In the case of C values, gdb provides exactly what Blink needs: the whatis command finds the type of an expression without executing it, and in particular, without causing any side effects or exceptions. In the case of Java values, jdb lacks the necessary functionality, so Blink emulates it using a simple work-around. Blink just needs to know whether it is dealing with a primitive value, such as a number, character, or boolean, or with a reference to a Java object or array. To discover this information, Blink instructs jdb to evaluate an expression that passes the value as a parameter to a helper method accepting java.lang.Object. If the call succeeds, the value is a reference. If the call has the wrong type, jdb will refuse to execute it, and will not cause any side effect. In this case, Blink catches the error message and treats the value as a primitive.

5.3 Expression Evaluation

This section explains each step of Blink's read-eval-print loop.

Reading. As suggested by Rosenberg, the "read" stage of Blink's REPL reuses syntax analysis code from a compiler [9]. But instead of just using the Java and C grammars, it reuses the grammar from Jeannie [5]. The Jeannie grammar is written in *Rats!*, a parser generator that uses packrat parsing for expressiveness and performance, and uses a module system to support scalable composition of parsers [4]. Jeannie's grammar is a composition of Java and C grammars, and Blink's expression grammar further augments it with user convenience variables, which look like identifiers starting with a dollar sign ($).

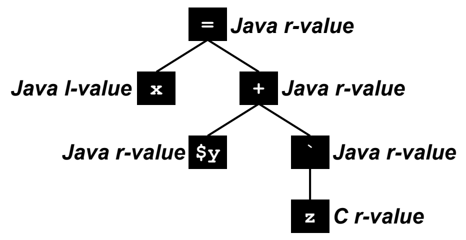

Figure 6. Reading the expression x = $y + `z when the current language is Java.

Instead of annotating the abstract syntax tree (AST) with types, Blink only annotates the AST with two pieces of information: the language (Java or C) of each node, and whether each node is an r-value (read-only) or an l-value (written-to on the left-hand side of an assignment). Figure 6 shows how Blink annotates the AST for the expression "x = $y + `z", assuming that the current language is Java. Node x is an l-value, since it is the left operand of the assignment =. Node z is C-language because `z`'s parent is the language toggle backtick `‘`.

Blink uses the component debuggers for symbol resolution. As is usual in debuggers, application symbols such as variable and function names are resolved relative to the current execution context. User convenience variables, on the other hand, have global scope and do not require context-sensitive lookup.

Evaluating. The interpreter visits the AST in depth-first left-to-right post-order. It is important that each node be executed exactly once and in the right order, to ensure the semantics in the presence of side effects, and to surprise users the least when an exceptional condition, such as a segmentation fault, cuts expression evaluation short.

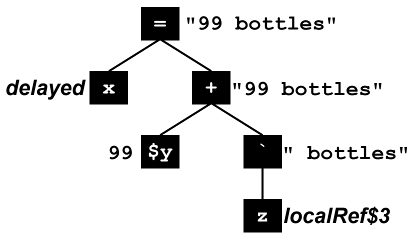

Figure 7. Evaluating the expression `x = $y + ‘z` when the current language is Java.

Figure 7 shows how Blink evaluates each AST node for the example code "`x = $y + ‘z`", assuming that the convenience variable `$y` currently stores the number `99`, and the C application variable `z` currently stores an opaque JNI local reference `localRef$3`. In this example, all leafs are variables, which Blink evaluates as described in Section 4.4. In general, leafs can also be literal values, which Blink uses directly without lookup. At inner nodes, Blink needs to perform evaluation actions. For the language toggle operator `‘` Blink performs a mixed-language data transfer as described in Section 5.2. For Figure 7, Blink discovers that the JNI reference `localRef$3` on the C side refers to the Java string `" bottles"` on the Java side. For other operators, such as `+` and `=` in Figure 7, Blink falls back on the REPL in the component debugger. Note that in general, an inner node may be a call to a user function and may thus execute arbitrary user code.

To evaluate an expression one AST node at a time, Blink needs temporary storage for subexpression results. For r-values, Blink evaluates the node, then stores the result in an internal convenience variable. For l-values, Blink evaluates their children, but delays their own evaluation. These l-values are evaluated later as part of their parent, which is by definition an assignment, such as `=` in Figure 7.

Printing. When expression evaluation reaches the root of the tree, Blink prints the result. As recommended by Rosenberg, Blink disables user breakpoints for the duration of expression evaluation, because the user would probably be confused and surprised when the expression hits a breakpoint in a callee [9]. But there may be other exceptional conditions during expression evaluation, such as a Java exception or a C segmentation fault. In this case, Blink aborts the evaluation of the current expression, and the debug session continues at the fault point instead. Whether expression evaluation terminates normally or aborts, Blink always nulls out internal convenience variables for subresults, and re-enables all user breakpoints.

6 Results

We evaluate Blink's scalability, functionality, portability, and performance and report our result here.

6.1 Scalability

Blink's implementation effort is scalable, because new code implements new functionality and existing functionality is borrowed from the single-language debuggers. To quantify this claim, we count non-blank and non-comment SLOC (source lines of code). Lines of code are an indirect metric for estimating the amount of effort to develop and maintain a software package.

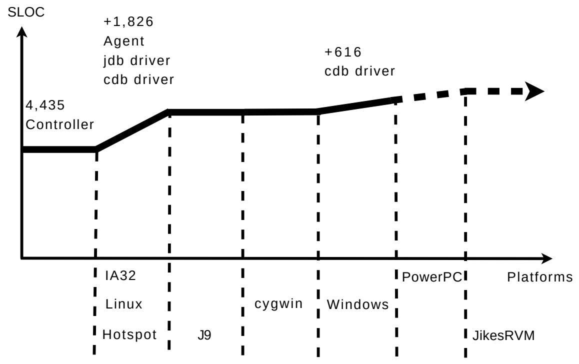

Figure 8. Scalable composition in Blink.

Figure 8 shows scalability of debugger composition in Blink. The basic composition framework costs 4,435 SLOC, and we added 1,826 SLOC to support our initial configuration: Sun Hotspot JVM running on the Linux/IA32 architecture. Once we paid this modest

1,826 SLOC, we did not need to implement any new features as we expanded the supported platforms such as IBM J9, cygwin, and Windows. We plan to add support for the Power PC architecture and for Jikes RVM and expect to need additional effort to handle the Power PC calling conventions.

For ideal scalable debugger composition, the graph in Figure 8 should be flat, since we did not implement any new features. However, there is a small additional cost, and there are two reasons. First, our native agent code contains small non-portable platform specific code to deal with the native call stack. Second, `cdb` exposes a different user interface than the richer `gdb`, and we implemented an adaptation layer to uniformly support either `gdb` on GNU platforms or `cdb` on Windows.

Debugger		SLOC	#Files
Blink		6,867	29
	Controller (front-end)	4,435	21
	Agent - java (back-end)	237	3
	Agent - C (back-end)	682	2
	jdb driver (back-end)	382	1
	gdb driver (back-end)	515	1
	cdb driver (back-end)	616	1
Java debugger - `jdb`		86,579	769
	jdb (user-interface)	18,360	122
	JDI (front-end)	16,983	256
	JDWP Agent (back-end)	40,171	356
	JVMTI (back-end)	11,065	35
C debugger - `gdb` 6.7.1		1,017,086	2,331
	gdb	419,921	1,524
	include	32,039	215
	bfd	286,981	398
	opcodes	278,128	194

Table 3. SLOC (source lines of code).

Table 3 shows the sizes of Blink, `jdb`, and `gdb` and their components. The `jdb` counts are for the `jdb` 1.6 source code in `demo/jpda/examples.jar` of the Sun JDK 1.6.0-b105. The JDI counts are for the JDI implementation in the Eclipse JDT. The JDWP and JVMTI counts are for corresponding subdirectories of the Apache DRLVM. Blink uses just 6,867 SLOC to effectively compose two existing Java and C debuggers. The SLOC of the existing debugger packages are 13 to 150 times larger than Blink's.

6.2 Functionality and Portability

We developed a number of primitive tests that cover the functionality of Blink. To demonstrate portability, we performed each of these test in different operating systems, JVMs, and with different C debuggers.

Context Management: This test sets two breakpoints, at `jPing` (`PingPong.java:7`) and `cPong` (`PingPong.c:17`) in Figure 1. During execution, the application stops at each of these breakpoints twice, and each time, the test uses the `backtrace` command.

Execution Control: This test first sets a breakpoint at the `main` method of the mutual recursion example in Figure 1. From there, the test repeatedly uses the `step` command until the end of the program. This exercises all cases of mixed-language stepping: both calls and returns, from both Java and C to the other language.

Data Inspection: This test first sets a breakpoint in a nested context of two example programs in the Blink regression test suite[1]. When the application hits the breakpoint, the test evaluates a variety of expressions, covering primitive and compound data, pure expressions and assignments, language transitions, and user function calls.

Results: Currently, all functionality tests succeed for the following configurations:

$$\left\{\begin{array}{l}\text{Sun JVM}\\ \text{IBM JVM}\end{array}\right\}+\left\{\begin{array}{l}\text{Linux}\\ \text{Cygwin}\end{array}\right\}+\texttt{gdb}$$

The "Cygwin" case is using Windows, but with the GNU C compiler instead of the Microsoft C compiler. We also tested Blink with Microsoft's C compiler and Microsoft's C debugger:

$$\left\{\begin{array}{l}\text{Sun JVM}\\ \text{IBM JVM}\end{array}\right\}+\text{Windows}+\texttt{cdb}$$

In these configurations, context management and execution control are fully supported, but data inspection is only partially supported, because `cdb`'s expression evaluation features are less powerful than those of `gdb`.

6.3 Performance

To measure the performance impact of Blink, we ran several large Java programs inside the debugger and measured runtimes. In Figure 9, we normalize it to the runtime with just `jdb` as a baseline, to capture any additional overheads that Blink might incur. The experiments use the JVM and `jdb` from Sun's JDK 1.6.0_03-b05 and `gdb` version 6.6. The initial heap size is 512MB, and the maximum heap size is 1GB. The experiments use a Pentium D 3.2GHZ running Linux 2.6.20. Each benchmark iterated once. The results are

[1]We omit the programs here for space reason.; the interested reader can find them in the open-source distribution of Blink, as `CompoundData.java/c` and `JeannieMain.jni`.

the median of 25 trials, because the adaptive optimizing compiler and garbage collector added some variation to the runtime.

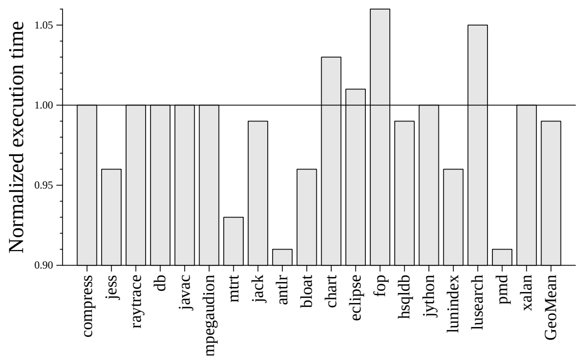

Figure 9. Blink's performance relative to `jdb`.

Figure 9 shows the results. Blink is 1% faster than `jdb` on average in spite of incurring various overheads (starting up and tearing down multiple single-language debuggers; running multiple processes; and adding a debugger agent inside the application, which overrides all C $\rightarrow$ Java transitions and adds two more indirections). The negligible overhead can be explained by the fact that the compositional debuggers are mostly inactive while the application runs at full speed. A counter-intuitive result is that some benchmarks appear to speed up a little; we believe that this slight anomaly is caused by locality effects or other interactions in the complicated software stack. Even with this performance anomaly, the results vary by only 9% across all benchmarks.

7 Conclusions

Debugging is one of the most difficult tasks in software development. It requires a knack for formulating the right hypotheses about bugs, and it requires the discipline to systematically verify or falsify hypotheses until the cause of the bug is found [14]. The analogy of a "dark cave" is apt, since debugging can be scary, and without the right tools, it can be a bumpy and aimless journey through twisted passages [9]. Single-language developers have long had good debugging tools to help them navigate the cave systematically. But mixed-language developers have been left in the dark, because it was difficult to write a good mixed-language debugger. We propose and evaluate a new way to build mixed-language debuggers more easily using scalable composition [7]. We use our insights to develop Blink, a debugger for Java and C. The open-source release of Blink is available as part of the `xtc` package:

`http://www.cs.nyu.edu/~rgrimm/xtc/`

Blink is full-featured and portable across different JVMs, operating systems, and C debuggers. Furthermore, Blink includes an interpreter (read-eval-print loop) for mixed-language expressions, thus providing users with a powerful tool not just for debugging, but also for testing, program understanding, and prototyping.

References

[1] JVM tool interface. `java.sun.com/javase/6/docs/technotes/guides/jvmti`, June 2006.

[2] P. Bothner. Compiling Java with GCJ. `http://www.linuxjournal.com/article/4860`, Jan. 2003.

[3] M. Chauvin, P. Ombredanne, and F. Granade. Support seamless debugging between JDT and CDT. `http://wiki.eclipse.org/Support_seamless_debugging_between_JDT_and_CDT`, Google Summer of Code project, 2007.

[4] R. Grimm. Better extensibility through modular syntax. In *Proc. 2006 PLDI*, pages 38–51, June 2006.

[5] M. Hirzel and R. Grimm. Jeannie: Granting Java native interface developers their wishes. In *Proc. 2007 OOPSLA*, pages 19–38, Oct. 2007.

[6] S. Liang. *The Java Native Interface: Programmer's Guide and Specification.* Addison-Wesley, June 1999.

[7] N. Nystrom, X. Qi, and A. C. Myers. J&: Nested intersection for scalable software composition. In *Proc. 2006 OOPSLA*, pages 21–36, Oct. 2006.

[8] V. Providin and C. Elford. Debugging native methods in Java applications. In *EclipseCon User Conference*, Mar. 2007.

[9] J. B. Rosenberg. *How Debuggers Work: Algorithms, Data Structures, and Architectures.* John Wiley & Sons, 1996.

[10] M. Stall. Mike Stall's .NET debugging blog. `http://blogs.msdn.com/jmstall/default.aspx`.

[11] Sun Microsystems. Debugging a Java application with dbx. `http://docs.sun.com/app/docs/doc/819-5257/blamm?a=view`, 2007.

[12] M. White. Debugging integrated Java and C/C++ code. `http://www.ibm.com/developerworks/java/library/j-jnidebug/index.html`.

[13] M. White. Integrated Java technology and C debugging using the Eclipse platform. In *JavaOne Conference*, Nov. 2006.

[14] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging.* Morgan Kaufmann, Oct. 2005.

[15] P. Zellweger. *Interactive source-level debugging.* PhD thesis, Xerox Parc Palo Alto Research Center, Technical Report CSL-84-5, 1984.