Debug All Your Code: Portable Mixed-Environment Debugging

Byeongcheol Lee
Martin Hirzel
Robert Grimm
Kathryn McKinley

OOPSLA 2009
Programmers build systems in multiple languages.

1. Leverage legacy code and existing libraries.
2. Match language features to a task.

**Portable mixed-environment debugging**

![Diagram showing mixed-language environments with FFI](image)
Programmers build systems in multiple languages.

1. Leverage legacy code and existing libraries.
2. Match language features to a task.
Bugs appear in all your code!
The problem with single-environment debugging

Eclipse JDT

Managed Java

Mixed-Environment
The problem with single-environment debugging
Our goal

Portable mixed-environment debugging

Managed Java

JNI

Native C/C++

Mixed-Environment
Our contribution and results

Composition

1. Add an intermediate agent.
3. Dispatch debuggers dynamically.

Blink results

1. **Simple**: Add 10 K SLOC of new code.
2. **Portable**: Support Linux, Windows, Hotspot, J9, GCC, Microsoft C++.
3. **Powerful**: Catch FFI bugs.
Problem: GDB does not work at a Java breakpoint.
Problem: GDB does not work at a Java breakpoint.

How can I print a C variable at a Java breakpoint? Unfortunately, GDB does not work at the Java breakpoint.
Our solution: the intermediate agent switches debugger context.

We will wake up GDB at a Java breakpoint - Switching debugger context.
Our solution: switch debugger context from Java to C.

```
j2c(){cbreak();
  void cbreak()
```

**Diagram:**
- JDB
- J-Agent
- C-Agent
- GDB

**Flow:**
- eval `j2c()`
- `j2c(){cbreak();`
- `void cbreak(){`
- Breakpoint hit
Our solution: switch debugger context from Java to C.

```
j2c(){
    eval j2c();
    cbreak();
}

void cbreak(){
    breakpoint hit
    print c_var
}
```
Our solution: switch debugger context from Java to C.

```c
void cbreak(){
    eval j2c();
    return;
}

return;
```

- **JDB**
- **J-Agent**
- **C-Agent**
- **GDB**

- j2c() completed
- eval j2c()
- void cbreak()
- return;
- return;
- breakpoint hit
- print c_var
- continue
Problem: GDB does not understand JNI calling conventions.

Naïve composition

JDB

GDB

How did the program reach the current breakpoint?

How can I tell you the calling context?

Unfortunately, GDB does not understand JNI transitions.
Problem: GDB does not understand JNI calling conventions.
Problem: GDB does not understand JNI calling conventions.

JDB stack:
- main
- jping(3)

GDB stack:
- cpong(2)

Diagram shows the stack frames with overlapping regions indicating the flow of control between `main`, `jping(3)`, and `cpong(2)`. The `STOP` symbol indicates a breakpoint or a stopping point in the execution.
Problem: GDB does not understand JNI calling conventions.
Problem: GDB does not understand JNI calling conventions.
Our solution: translate JNI transitions into C transitions.

The agent translates JNI calls into C calls – transition interposition.
Our solution: compose a calling context.
Our solution: controller and intermediate agent

- Controller
- JDB
- Managed Java agent
- GDB
- Native C/C++
- Intermediate agent
I. Problem
II. Debugger composition
   A. Switching debugger context
   B. Interposing transitions
III. Advanced features
   A. Evaluating Jeannie mixed-environment expressions
   B. Detecting FFI bugs
IV. Evaluation
Debugging boundary code

What do I need to debug boundary code?

Managed Java

JNI

Native C/C++

Mixed-Environment
1. Evaluating Jeannie Expressions

2. Detecting FFI bugs

Advanced features to debug boundary code
Problem: you cannot de-reference opaque pointers in C.

```c
jstring name;
```
Problem: you can not de-reference opaque pointers in C.

print *name

The name is an opaque C pointer.
Our solution: use Jeannie expression

```
print "smiley"

jstring name;
```

The `name` is an opaque C pointer.

The Jeannie backtick gives me eyes to see the Java world.
B. Lee, M. Hirzel, R. Grimm, and K. S. McKinley

Our solution: use Jeannie expression.

```
name;
print *name
print `name
```

The name is an opaque C pointer.

How can I evaluate the Jeannie expression?
Build abstract syntax tree

print

\`
name
\`
Evaluate AST in Bottom-up order.

print

\$
vc_1 = \text{name}
\$

GDB

Agent

JDB

name
Evaluate AST in Bottom-up order.

- GDB
  \[ \text{\$vc}_1 = \text{name} \]

- Agent
  \[ \text{\$vj}_2 = \text{\$vc}_1 \]

- JDB

- print
  \`name\`
Evaluate AST in Bottom-up order.

```
print `name

$vc_1 = \text{name}

$vj_2 = \$vc_1

Print $vj_2

"smiley"
```
Advanced features

A. Evaluating Jeannie Expressions

B. Detecting FFI bugs
Problem: you may misuse foreign function interface.

C code

(*env)->GetStringUTFChars(env, NULL);
The **NULL** is invalid.
The java-gnome bug 576107 **crashes** the J9 JVM.

C code

```c
(*env)->GetStringUTFChars(env, NULL);
```
Our solution: detect FFI bugs

The **NULL** is invalid.
The java-gnome bug 576107 **crashes** the IBM J9 SR5.

```c
(*env)->GetStringUTFChars(env, NULL);

c2j_wrap_GetStringUTFChars(env, cstr) {
```
Our solution: detect FFI bugs.

The null is invalid.
The java-gnome bug 576107 crashes the IBM J9 SR5.

```c
C code

(*env)->GetStringUTFChars(env, NULL);

c2j_wrap_GetStringUTFChars(env, cstr) {

if (cstr == NULL) {cbreak();
```
Our solution: detect FFI bugs.

The `NULL` is invalid. The java-gnome bug 576107 crashes the IBM J9 SR5.

The agent immediately reports the bug.
I. Problem

II. Debugger composition
   A. Switching debugger context
   B. Interposing transitions

III. Advanced features
   A. Evaluating Jeannie mixed-environment expressions
   B. Detecting FFI bugs

IV. Evaluation
Composition is simple.

515K source lines of code in total
Composition is portable.

Source lines of code

Platforms

IA32/Linux/ Hotspot

Agent

Controller

8,840

4,575

4,575

0
Composition is portable.

![Graph showing source lines of code and platforms]

- IA32/Linux/Hotspot: 4,575
- J9: 8,840

- Agent
- Controller
Composition is portable.
Composition is portable.

Platforms

IA32/Linux/Hotspot  J9  Cygwin  Windows

Source lines of code

0  4,575  8,840  9,481

Agent  Controller
Composition is powerful.

The bug 576107 in java-gnome 4.0.10

<table>
<thead>
<tr>
<th></th>
<th>Hotspot VM 1.6.0_10</th>
<th>J9 VM SR5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Production run</td>
<td>running</td>
<td>crash</td>
</tr>
</tbody>
</table>
Composition is powerful.

The bug 576107 in java-gnome 4.0.10

<table>
<thead>
<tr>
<th></th>
<th>Hotspot VM 1.6.0_10</th>
<th>J9 VM SR5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Production run</td>
<td>running</td>
<td>crash</td>
</tr>
<tr>
<td>Runtime checking (-Xcheck:jni)</td>
<td>warning</td>
<td>warning</td>
</tr>
</tbody>
</table>
The bug 576107 in java-gnome 4.0.10

<table>
<thead>
<tr>
<th></th>
<th>Hotspot VM 1.6.0_10</th>
<th>J9 VM SR5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Production run</td>
<td>running</td>
<td>crash</td>
</tr>
<tr>
<td>Runtime checking</td>
<td>warning</td>
<td>warning</td>
</tr>
<tr>
<td>(-Xcheck:jni)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jdb</td>
<td>running</td>
<td>crash</td>
</tr>
<tr>
<td>gdb</td>
<td>running</td>
<td>fault</td>
</tr>
</tbody>
</table>
The bug 576107 in java-gnome 4.0.10

<table>
<thead>
<tr>
<th></th>
<th>Hotspot VM 1.6.0_10</th>
<th>J9 VM SR5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Production run</td>
<td>running</td>
<td>crash</td>
</tr>
<tr>
<td>Runtime checking (-Xcheck:jni)</td>
<td>warning</td>
<td>warning</td>
</tr>
<tr>
<td>gdb</td>
<td>running</td>
<td>crash</td>
</tr>
<tr>
<td>gdb</td>
<td>running</td>
<td>fault</td>
</tr>
</tbody>
</table>

**Blink**

breakpoint
breakpoint
Related work

• Mixed-environment debuggers
  – Intel XDI for Harmony JVM
  – SUN dbx
  – Microsoft .NET debuggers

• Advance debugging features
  – Static analyses
    • BEAM [Kondoh & Onodera ’08]
    • J-Saffire [Furr & Foster ’06]
    • ILEA [Tan & Morrisett ‘07]
  – Language designs
    • Jeannie [Hirzel & Grimm ‘07]
    • SafeJNI [Tan et al. ‘06]
  – Wrapper generators
    • Automatic binding generator [Ravitch ‘09]
    • SWIG [Beazley ‘96]
Summary

• Portable mixed-environment debugging
• Composition with an intermediate agent
  1. Switching debugger context
  2. Interposing transitions
• Results
  1. Simple
  2. Portable
  3. Powerful
Thank you