Reverse Execution of Java Bytecode

JONATHAN J. COOK

Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh EH9 3JZ, UK
Email: Jon.Cook@ed.ac.uk

We demonstrate a model, including operational semantics, for the reverse execution of stack-based code. We discuss our modification of the Kaffe implementation of the Java Virtual Machine, supporting a debugger capable of running Java bytecode backwards. We achieve reverse execution by logging the state lost during each operation or by directly reversing instructions. Our debugger has facilities for stepping, stepping over methods and running to breakpoints, in both directions. Multi-threading is supported. It is also possible to step through the bytecode when the Java source code is not available. The debugger has both a command line user interface and a graphical user interface with facilities for editing code and running the Java compiler.

Received 30 November 2001; revised 13 May 2002

1. INTRODUCTION

There are many debuggers available for debugging Java programs. For instance, Sun's JDK includes a command-line-based debugger called JDB [1]. Various graphical debugging environments have been developed, for instance Borland JBuilder [2] and Forte for Java [3]. NetBeans [4], which incorporates a debugger for Java, is open-source and written in Java. Such debuggers have facilities for stepping forwards through the source code a line at a time and inspecting the values of variables during the execution of the program being debugged.

Most commercially available debuggers are only capable of stepping forwards through the source code. Because the location of the bug is usually unknown, we may find after having stepped through hundreds of lines of code that we have stepped well past the bug and have to start again. In such instances it can be helpful to be able to step back through the source code to the location of the bug.

We discuss our implementation of a debugger which is capable of running Java bytecode backwards. This allows us to jump to a point in the code which we know to be after the bug and then step backwards until the bug is found. This is particularly useful when debugging large or long running programs. In addition, it is useful for correcting forward steps made in error, which is a problem that frequently occurs while single-stepping through programs.

There are a few possible approaches to implementing reverse execution. One is to record the data necessary for reverse execution as the program runs forwards, which we refer to as 'logging'. Another is to use checkpoints, see [5] and [6]. With checkpointing, the entire execution state of the program is periodically stored in memory or written to disk. Then, as in the case of [6], to step backwards one line we go to the last checkpoint and then run forwards until the line just before the one we started on. A third approach is to convert the code into inherently reversible code or, as in the case of [7], directly reverse instructions. Our debugger uses logging as the main approach.

To implement reverse execution one can either instrument the code or modify the compiler or interpreter. With instrumentation the debugger rewrites the source code to incorporate support for reverse execution and then recompiles the code and runs it, without any modification to the compiler or interpreter. Our approach is to modify the interpreter to support controlled execution.

In implementing our debugger, we demonstrate an efficient method of reversing stack-based code by taking advantage of the stack model. We also demonstrate a method for reversing methods calls, returns from methods and exceptions in the case of the Java Virtual Machine (JVM), Kaffe. Kaffe is structured so that it simulates a method call in the Java code by a function call in its own code. To achieve reverse execution we pass extra data to and return extra data from the function in Kaffe which runs the bytecode of a method.

In this paper, we first motivate and describe the data structure that we use for logging and discuss the implementation of reverse execution. We then describe how we deal with method calls, exceptions, garbage collection, multi-threading and file I/O, and give some details of the architecture of the tool. Finally, we include a brief case study, give performance data and discuss related work.

2. DATA STRUCTURE FOR REVERSE EXECUTION

2.1. Characterization of bytecode instructions

We wish to tailor the reverse execution scheme to a stack-based virtual machine. Almost all of the Java bytecode instructions can be characterized by two numbers, m and n, where the instruction can be thought of as popping m items from the operand stack, possibly operating on them and then pushing n new items onto the operand stack. In some cases the entire operand stack is popped. It is an easy, if laborious, exercise to determine such numbers. See the JVM specification [8] for detailed descriptions of the bytecode instructions.

In order to reverse the bytecode instructions, we need to somehow store the *m* items of the operand stack that are lost when running the instruction forwards, so that they can be restored on reverse execution.

The instruction *IADD* takes two integers off the operand stack, adds them together, and then pushes the sum back onto the operand stack. Thus, in this case $m = 2$ and $n = 1$. When running forwards we must record the top two values of the operand stack before the instruction executes as usual. On reversing we must discard the top value of the operand stack before restoring the two values that we recorded earlier.

In view of the possibility of branches and jumps we also need to store data on the flow of control, that is the order in which the instructions are executed.

We store these data on circular buffers.

2.2. Two circular buffers

A circular buffer is a bounded stack with the property that if we push beyond the end, we wrap around and begin overwriting the start of the underlying array. We use two circular buffers to record the data that is necessary for reverse execution. The circular buffers employ a stack-like last-in-first-out discipline which makes them similar, for example, to the kill ring of the Emacs text editor.

Before we execute each instruction, we push its program counter value onto the program counter circular buffer. As part of the forwards execution of an instruction, we push any data which will be lost as a result of the instruction onto the other circular buffer, which we refer to as the logging circular buffer.

By default, the program counter circular buffer is 128×1024 slots and the logging buffer is also 128×1024 slots. In total this equates to about 2 Mbytes of memory. This figure includes the other buffers in parallel with the logging circular buffer, which we discuss below. The user is able to set the size of the buffers. For some of the sample programs we tried, we found that on average 8 buffer slots of each buffer were used for each line of source code. Thus, on average we can reverse through 16,000 lines of code, with the default setting.

2.3. Issues arising from the boundedness of the buffers

However, how do we know whether we can step back a line or whether the data for the line to step back through has been overwritten because the buffer has wrapped around? We do this by keeping a validity pointer, marking the furthest back we can read.

Before a backwards step is effected, we make sure that we have enough data on the buffers to go back. We do not want to find part way through reversing an instruction that we have run out of data. So, before stepping back, we run a simulation of the reverse step using code similar to that which performs the real reversal. In terms of flow of control this is relatively straightforward but a problem arises with the logging circular buffer. We do not want to actually reverse the instructions, but we need to know how many items to take off the logging circular buffer for each instruction. We do this by adding an extra circular buffer, parallel to the logging circular buffer. The elements of this buffer are initialized to zero. Before each instruction, the element corresponding to the current position is incremented. Then to run back over the last instruction, we go back skipping those entries which are zero until we reach a non-zero value which we decrement.

The following simplified code should convey the idea. Before the instruction begins, `mark()` is called. Then `push()` will be called some number of times or not at all. When reaching this point, on simulating a reversal, `skipBack()` will remove exactly one instruction's data from the buffer, regardless of whether that instruction pushed to the buffer.

```
Procedure mark( ) {
   increment( count[location] );
}

Procedure push( value ) {
   log[location] = value;
   increment( location );
   if( location == BUFFER_SIZE )
     location = 0;
}

Procedure skipBack( ) {
   while( count[location] == 0 )
      decrement( location );
   decrement( count[location] );
}
```

The reader may feel that this scheme introduces unnecessary complexity and that if the buffer runs out in the middle of a reversal we should simply run the code forwards again to where we should be. We cannot, however, do this as not all native methods are reversed. Thus, if we were unable to step back over an instruction which wrote to the console, this instruction would be replayed and extra output would appear.

3. IMPLEMENTATION OF REVERSE EXECUTION

We describe the reversal of instructions in two ways. In this section we explain the reversal in prose. In the next section we give operational semantics for both directions of execution.

It should be noted that two of the basic types which can be stored on the operand stack (longs and doubles) actually take up two slots. In the descriptions and operational semantics, we regard such pairs as single stack slots.

3.1. Description of reversal without method calls

3.1.1. The operand stack and object creation

Recall the values m and n that we attributed to some bytecode instructions above. We reverse most instructions (roughly 150 of the 200 bytecode instructions) as follows. When the instruction is reached, we push the top m items of the operand stack onto the logging circular buffer. When the instruction is reversed, we pop and discard the top n items of the operand stack and then push the top m items of the logging circular buffer back onto the operand stack. This reverses the instruction. The instructions which can be reversed in this way are those which can be treated as though they operated only on the operand stack. This class of instructions is referred to as **STACK** in the semantics.

MULTIANEWARRAY is reversed in a way similar to the **STACK** scheme, except that the number of items of the operand stack to be saved depends on a field of the bytecode instruction, rather than the instruction itself.

We do not need to do anything additional in order to reverse the *NEW* instructions, as they generate new references on each forwards pass, which can be used thereafter. The constructor is called by an *INVOKESPECIAL* instruction after the *NEW*. When this instruction is reached the constructor is stepped through and logged.

3.1.2. Local variables

Local *STORE*s take a value off the operand stack and store it in a local variable. We reverse these as follows. When we execute in the forwards direction, we push the old value of the local variable onto the logging circular buffer, before copying the value on the top of the operand stack into the local variable. To reverse this, we move the local variable value back to the operand stack and then the value saved on the logging circular buffer back to the local variable.

IINC increments a local variable by a specified amount. We reverse *IINC* directly, by subtracting the relevant value rather than adding it.

WIDE instructions are used when there are very many local variables and a wider index into the local variable table is needed; or in the case of *IINC*, a large increment is required. We reverse each *WIDE* instruction in the same way as its non-*WIDE* counterpart.

3.1.3. Fields

We have to reverse array *STORE*s taking into account the need for the second and third items from the top of the operand stack when locating the address to be stored into. Thus, in the forwards direction, we push onto the logging circular buffer: the old value of the array element together with the second and third items of the operand stack. We then copy the value on the top of the operand stack into the array. To reverse this, we load the values from the logging circular buffer onto the operand stack. We use these to locate the array element and to copy its value onto the operand stack. We then copy the old saved value on the logging circular buffer into the array.

GETSTATIC loads a static field, pushing it onto the operand stack. We reverse this by popping off the top value of the operand stack.

We reverse *PUTSTATIC* in a way similar to the local stores, except that we are dealing with a field instead of a local variable.

GETFIELD loads a non-static field onto the operand stack. We reverse *GETFIELD* as follows. In the forwards direction we push the object reference on the top of the operand stack onto the logging circular buffer, then we get the field in the usual manner. We reverse this by popping the top value off the operand stack and then loading the top item of the logging circular buffer onto the top of the operand stack.

We reverse *PUTFIELD* in a manner similar to the array stores, as here we need the object reference on the operand stack to locate the field.

3.1.4. Branches, subroutines and jumps

Since the program counter value of every instruction is stored on the program counter buffer, no special treatment of branches, subroutines or jumps is required. These instructions are all dealt with as part of the **STACK** scheme. As the program counter often merely increases by one, some space could be saved by compressing the data on the program counter circular buffer. However, this may introduce an unacceptable time overhead.

3.2. Reversing method transitions

We now consider the reversal of *RETURN*s and *INVOKE*s. That is, we reverse all those instructions which result in transitions between methods. In [7], Biswas and Mall discuss the reversal of function calls. They observe that when a function exits, its stack frame and local variables are lost. Hence, these values need to be stored before leaving the function, so that they can be restored when stepping back into the end of the function.

3.2.1. Method transitions: running forward normally

A method call consists of pushing the actual arguments and, if necessary, the object of the method to be called onto the operand stack. Then one of the *INVOKE* instructions is called. This pops these values from the operand stack and a C function is called in the Kaffe JVM, which runs the Java method being called. On entry into the new method, space is allocated for a new operand stack and local variables. This space is discarded on exiting from the method. The operand stacks of the calling and called methods are separate and exist as local variables of the respective function calls in the Kaffe JVM.

Thus, in order to be able to reverse method calls, we must save the arguments before the method call and save the operand stack and local variables just before the method returns.

3.2.2. *Method transitions: running forward modified*

When a method is called, we pass an extra parameter to the corresponding function call in the Kaffe virtual machine. This value indicates whether to run the method forwards or backwards and whether to step into or over the method. As discussed below, there are actually two 'forwards into' values and other values used when reversing an exception.

We also modify the function in Kaffe which runs the bytecode for a method to return a value indicating whether the method exited from the beginning or the end. The method can also return values which indicate that an exception is being reversed.

We store tagged values on the program counter circular buffer. The program counter is in the range `0x0` to `0xFFFF`, as the maximum size of any method in a Java class file is `0x10000` bytes; see [8, Section 4.10]. The program counter circular buffer stores 32-bit integers so that we can use larger integers to store extra data. We define

$$\begin{aligned} \mathit{start} &= \texttt{0x20000} \\ \mathit{end} &= \texttt{0x20001} \\ \mathit{offset} &= \texttt{0x10000}. \end{aligned}$$

At the start of a method, if it is being logged, we push *start* to the program counter circular buffer and similarly *end* at the end.

When we call a method running forwards, the relevant *INVOKE* is first called. This is modified to save the top portion of the operand stack comprising of the arguments and, where necessary, the relevant object to the logging circular buffer. In the cases of *INVOKESPECIAL*, *INVOKEVIRTUAL* and *INVOKEINTERFACE*, to do the reverse call we also need the object reference, so a copy of it is made to push onto the logging circular buffer later. The method is then called.

The method returns when execution reaches one of the *RETURN* instructions. Before returning, we push the entire operand stack and all of the local variables onto the logging circular buffer. Usually at a point of return, the operand stack is almost empty; however, there is a significant overhead if there are many local variables. Fortunately, in realistic code this is uncommon.

After the method returns, if it exits at the end and if appropriate, we push the object reference which we recorded earlier onto the logging circular buffer. Then, only if the method exits at the end, we push the value of the program counter of the *INVOKE* instruction with *offset* added to it, that is `pc + 0x10000`, onto the program counter circular buffer. If the method exits out of the beginning, we jump to the reverse execution code in the function call in Kaffe running the Java calling method.

3.2.3. *Reverse execution of method calls*

In order to step back, we pop values off the two circular buffers as appropriate. If we hit a *start* then we return the value indicating that we have stepped out of the beginning of the method.

If we hit a value on the program counter circular buffer between `0x10000` and `0x1FFFF`, we know that we have to reverse a method call and we look up the value `pc & 0xFFFF` to find the *INVOKE* instruction needed to call the method in reverse.

On entry into a method backwards, we first encounter one of the *RETURN* instructions, which restores the local variables and the operand stack. We then continue until we reach the beginning and exit, or change direction and reach the end of the method, in which case the calling function transfers control to the forwards execution code in the function call running the Java calling method, after the return.

3.3. Exceptions

When an exception occurs, either caused by an *ATHROW* instruction or the failure of some other instruction, the JVM looks up the call stack for a handler and then, in the case of Kaffe, uses a C long jump to jump into the handler.

Before an exception is dispatched, our modified code looks up the call stack until it reaches the handler, pushing onto the logging circular buffer the operand stacks and local variables of every logged method it sees. We also note on the program counter circular buffer the program counter value of the instruction which faulted and the program counter value of each *INVOKE* instruction in the call chain.

To reverse an exception, we first call the method which caught the exception and jump straight to the instruction which called the next method in the chain, having restored the operand stack to the state that it was in just before that call originally happened. We continue doing this until we reach the method where the exception was thrown. Here we restore the operand stack and local variables and then run back to the beginning of the line containing the instruction which faulted.

In some cases we could step back further and provide different input, which would mean that on the second pass the exception would not occur and the method would terminate normally. In this instance it is important that when the method returns, the method which called it continues as it would normally. This is why when reversing we do not simply call the method that threw the exception.

The implementation of exceptions relies on extra flags being passed to and returned from the function which runs the bytecode. These have the effect of causing the current method to be re-run starting at a different program counter value. We want to do this because when an exception in Kaffe is caught in a method, rather than staying in the method and jumping to the catch block's program counter value, the method is re-run with the program counter set to the appropriate location. Further values are required for the program counter circular buffer; see the final rule of Figure 4.

4. OPERATIONAL SEMANTICS

4.1. Operational semantics without method calls

We now describe the notation used in the operational semantics. We will extend the notation to cater for method calls in the next section. To give operational semantics for the full Java language would be a tremendous task, so we abstract away, in subsidiary functions, all those parts of it which are not relevant to reverse execution.

We indicate the target program by the symbol P, and the state by an ordered sextuple $\langle pc, S, L, F; D, C\rangle$. The items before the semicolon are the program counter (pc), the operand stack (S), the local variables (L) and the object/class fields (F). The items after the semicolon are added to enable reverse execution and are the logging circular buffer (D) and the program counter circular buffer (C).

The judgement $P \vdash \langle pc, S, L, F; D, C\rangle \longleftrightarrow \langle pc', S', L', F'; D', C'\rangle$ indicates that the program running in the forwards direction can make the transition indicated by going to the right in one step. The program running in the reverse direction can make the transition indicated by going to the left in one step. Premises enclosed in square brackets are only required in the forwards direction.

4.1.1. *Special functions*

$P[pc]$ denotes the bytecode instruction at the program counter location pc, with the bytecode arguments of the instruction written after the instruction name. We refer to sets of instructions and in an abuse of notation, add the bytecode arguments after these as well. For example, **LOCALSTORES** l indicates instructions which are a member of the set **LOCALSTORES** and which operate on the local variable l.

$pops[P[pc]]$ and $pushes[P[pc]]$ are the values m and n, respectively, discussed above. $nextpc(P[pc], pc)$ returns the value that the program counter should change to after executing the instruction at program counter value pc, in the forwards direction.

We use the function *trans* to hide most of the aspects of the JVM which are not relevant to reverse execution. It takes as arguments the instruction, program counter, operand stack and current local variables ($P[pc], pc, S, L$) and returns a triple $\mathbf{norm}(pc', S', F^+)$ consisting of the new program counter, the new operand stack and any new fields to add. F^+ is a set and hence can take the value $\varnothing$ indicating that no new fields are added. We need to obtain the new program counter in this way, rather than by *nextpc*, because with some instructions the next program counter value depends on the operand stack. *trans* can also return the value $\mathbf{exn}(pc_{catch}, n, o)$, which indicates that the instruction threw an exception. Here, pc_{catch} is the program counter value of the catch clause in the catching method. n is the number of methods we are to jump back through and o is the object representing the exception.

Java integer addition is represented by the symbol $+_J$.

4.1.2. *States*

We denote the result of pushing the value x onto the stack S by $x \cdot S$. We denote the empty stack by ϵ and the result of appending stack S' onto stack S by $S' \bullet S$. The expression $|S|$ denotes the number of elements in the stack S.

If $L[l = r]$ occurs on the left of a transition and $L[l = r']$ occurs on the right, this indicates that the value of the local variable l has changed from r to r', all other values remaining the same.

A similar notation is used for the fields, F. The domain of F is the set of all fields that can exist in the target program, which consists of the static fields of each class and non-static fields of each object. Static fields are represented by a single value i and non-static fields are represented by a pair (o, i) consisting of the object o and field number i. The ith element of the array a is represented by $a[i]$. The range of F is all Java values and references. The instruction *NEW* is classified under **STACK**, but in addition to generating a new object reference, it creates a new object together with all of its fields. These constitute the F^+ in the rule for **STACK**. The rule might appear to suggest that these extra fields disappear when we reverse a *NEW*. This will only occur, however, on a later garbage collection cycle.

We do not fully represent the effects of possible garbage collection on F (see Section 6), nor of the acquisition and release of monitors (see Section 7.1). Both of these would unnecessarily complicate the semantics.

D and C are circular buffers, and we indicate the result of pushing the value x onto the buffer D by $x \cdot D$ and similarly for C. Again we use the symbols ϵ and $\bullet$ for the empty buffer and for appending a stack onto a buffer. The push here is subtly different to that for S, in that if we push many items onto D or C, then items begin to disappear from the other end of the buffer.

The initial state is $\langle 0, \epsilon, L, F; \epsilon, \epsilon\rangle$, where L and F contain initial values for the local variables and fields.

4.1.3. *The rules*

We use similar notation to that used in [9] and [10] which give operational semantics for a small version of the JVM. The reader may like to inspect these before looking at our operational semantics.

In Figure 1, we classify the JVM instructions. In Figure 2, we give the operational semantics without method calls or exceptions. In Figure 3, we give an example instantiation of the **STACK** rule.

4.2. Operational semantics of method calls and exceptions

We now add some notation. The sextuple is replaced by a septuple $\langle Q, M, \mathbb{S}, \mathbb{L}, F; D, C\rangle$. Q indicates a stack of program counter values and M a stack of method descriptors, with the symbols $\cdot$ and ϵ used as before. F, D and C are notationally unchanged.

Now, however, $\mathbb{S}$ and $\mathbb{L}$ indicate stacks of stacks. The result of pushing the stack S onto the stack of stacks $\mathbb{S}$ is denoted by $S \odot \mathbb{S}$ and the empty stack of stacks is denoted $[\,]$. This is a notational convenience to enable us to simply denote the popping of entire stack frames when we exit from a method.

The initial state is $\langle 0 \cdot \epsilon, main \cdot \epsilon, \epsilon \odot [\,], L \odot [\,], F; \epsilon, \epsilon\rangle$. Here, L consists of the arguments to the Java application;

$$\begin{aligned}
\textbf{INSTRUCTIONS} &= \text{the set of Java bytecode instructions}\\
\textbf{LOCALSTORES} &= \{\mathit{ISTORE}, \mathit{LSTORE}, ..., \mathit{ASTORE}\}\\
\textbf{ARRAYSTORES} &= \{\mathit{IASTORE}, \mathit{LASTORE}, ..., \mathit{AASTORE}\}\\
\textbf{FIELDS} &= \{\mathit{GETSTATIC}, \mathit{PUTSTATIC}, \mathit{GETFIELD}, \mathit{PUTFIELD}\}\\
\textbf{STATICINVOKES} &= \{\mathit{INVOKESTATIC}\}\\
\textbf{OBJECTINVOKES} &= \{\mathit{INVOKEVIRTUAL}, \mathit{INVOKEINTERFACE}, \mathit{INVOKESPECIAL}\}\\
\textbf{RETURNS} &= \{\mathit{IRETURN}, \mathit{LRETURN}, ..., \mathit{RETURN}\}\\
\textbf{MONITOR} &= \{\mathit{MONITORENTER}, \mathit{MONITOREXIT}\}
\end{aligned}$$

$$\begin{aligned}
\textbf{STACK} = {} & \textbf{INSTRUCTIONS} \setminus (\,\textbf{LOCALSTORES} \cup \textbf{ARRAYSTORES}\\
& \cup \textbf{FIELDS} \cup \textbf{STATICINVOKES} \cup \textbf{OBJECTINVOKES} \cup \textbf{RETURNS}\\
& \cup \textbf{MONITOR} \cup \{\mathit{IINC}, \mathit{WIDE}, \mathit{MULTIANEWARRAY}, \mathit{ATHROW}\}\,)
\end{aligned}$$

FIGURE 1. Classes of instruction.

Stack:

$$\frac{\begin{array}{c} P[pc] \in \textbf{STACK} \quad pops[P[pc]] = m \quad pushes[P[pc]] = n \\ [\mathit{trans}(P[pc], pc, x_1 \cdot x_2 \ldots x_m \cdot S, L) = \textbf{norm}(pc', y_1 \cdot y_2 \ldots y_n \cdot S, F^+)] \end{array}}{P \vdash \langle pc, x_1 \cdot x_2 \ldots x_m \cdot S, L, F;\, D, C\rangle \longleftrightarrow \langle pc', y_1 \cdot y_2 \ldots y_n \cdot S, L, F \cup F^+;\, x_1 \cdot x_2 \ldots x_m \cdot D, pc \cdot C\rangle}$$

$$\frac{P[pc] = \mathit{MULTIANEWARRAY}\ m \quad [\mathit{trans}(P[pc], pc, d_1 \cdot d_2 \ldots d_m \cdot S, L) = \textbf{norm}(pc', o \cdot S, F^+)]}{P \vdash \langle pc, d_1 \cdot d_2 \ldots d_m \cdot S, L, F;\, D, C\rangle \longleftrightarrow \langle pc', o \cdot S, L, F \cup F^+;\, d_1 \cdot d_2 \ldots d_m \cdot D, pc \cdot C\rangle}$$

Locals:

$$\frac{P[pc] \in \textbf{LOCALSTORES}\ l \quad [\mathit{nextpc}(P[pc], pc) = pc']}{P \vdash \langle pc, v \cdot S, L[l = x], F;\, D, C\rangle \longleftrightarrow \langle pc', S, L[l = v], F;\, x \cdot D, pc \cdot C\rangle}$$

$$\frac{P[pc] = \mathit{IINC}\ l\ n \quad [\mathit{nextpc}(P[pc], pc) = pc']}{P \vdash \langle pc, S, L[l = x], F;\, D, C\rangle \longleftrightarrow \langle pc', S, L[l = x +_J n], F;\, D, pc \cdot C\rangle}$$

Fields:

$$\frac{P[pc] \in \textbf{ARRAYSTORES} \quad [\mathit{nextpc}(P[pc], pc) = pc']}{P \vdash \langle pc, v \cdot i \cdot a \cdot S, L, F[a[i] = x];\, D, C\rangle \longleftrightarrow \langle pc', S, L, F[a[i] = v];\, x \cdot i \cdot a \cdot D, pc \cdot C\rangle}$$

$$\frac{P[pc] = \mathit{GETSTATIC}\ i \quad [\mathit{nextpc}(P[pc], pc) = pc']}{P \vdash \langle pc, S, L, F[i = x];\, D, C\rangle \longleftrightarrow \langle pc', x \cdot S, L, F[i = x];\, D, pc \cdot C\rangle}$$

$$\frac{P[pc] = \mathit{PUTSTATIC}\ i \quad [\mathit{nextpc}(P[pc], pc) = pc']}{P \vdash \langle pc, v \cdot S, L, F[i = x];\, D, C\rangle \longleftrightarrow \langle pc', S, L, F[i = v];\, x \cdot D, pc \cdot C\rangle}$$

$$\frac{P[pc] = \mathit{GETFIELD}\ i \quad [\mathit{nextpc}(P[pc], pc) = pc']}{P \vdash \langle pc, o \cdot S, L, F[(o, i) = x];\, D, C\rangle \longleftrightarrow \langle pc', x \cdot S, L, F[(o, i) = x];\, o \cdot D, pc \cdot C\rangle}$$

$$\frac{P[pc] = \mathit{PUTFIELD}\ i \quad [\mathit{nextpc}(P[pc], pc) = pc']}{P \vdash \langle pc, v \cdot o \cdot S, L, F[(o, i) = x];\, D, C\rangle \longleftrightarrow \langle pc', S, L, F[(o, i) = v];\, x \cdot o \cdot D, pc \cdot C\rangle}$$

FIGURE 2. Operational semantics without method calls.

$$\frac{\begin{array}{c} P[pc] = \mathit{IADD} \in \textbf{STACK} \quad pops[\mathit{IADD}] = 2 \quad pushes[\mathit{IADD}] = 1 \\ [\mathit{trans}(\mathit{IADD}, pc, x_1 \cdot x_2 \cdot S, L) = \textbf{norm}(pc + 1, (x_1 +_J x_2) \cdot S, \varnothing)] \end{array}}{P \vdash \langle pc, x_1 \cdot x_2 \cdot S, L, F;\, D, C\rangle \longleftrightarrow \langle pc + 1, (x_1 +_J x_2) \cdot S, L, F;\, x_1 \cdot x_2 \cdot D, pc \cdot C\rangle}$$

FIGURE 3. Example: **STACK** rule instantiated to *IADD* instruction.

Invokes:

$$\frac{P[pc, m] \in \textbf{STATICINVOKES}\ m' \quad [\mathit{newlocals}(m', x_1 \ldots x_n \cdot \epsilon) = L']}{\begin{array}{c} P \vdash \langle pc \cdot Q, m \cdot M, x_1 \cdot x_2 \ldots x_n \cdot S' \odot \mathbb{S}, \mathbb{L}, F;\ D, C\rangle \\ \longleftrightarrow \langle 0 \cdot pc \cdot Q, m' \cdot m \cdot M, \epsilon \odot S' \odot \mathbb{S}, L' \odot \mathbb{L}, F;\ x_1 \cdot x_2 \ldots x_n \cdot D, \mathit{start} \cdot m \cdot pc \cdot C\rangle \end{array}}$$

$$\frac{P[pc, m] \in \textbf{OBJECTINVOKES}\ m' \quad [\mathit{newlocals}(m', x_1 \ldots x_n \cdot o \cdot \epsilon) = L']}{\begin{array}{c} P \vdash \langle pc \cdot Q, m \cdot M, x_1 \cdot x_2 \ldots x_n \cdot o \cdot S' \odot \mathbb{S}, \mathbb{L}, F;\ D, C\rangle \\ \longleftrightarrow \langle 0 \cdot pc \cdot Q, m' \cdot m \cdot M, \epsilon \odot S' \odot \mathbb{S}, L' \odot \mathbb{L}, F;\ x_1 \cdot x_2 \ldots x_n \cdot o \cdot D, \mathit{start} \cdot m \cdot pc \cdot C\rangle \end{array}}$$

Returns:

$$\frac{P[pc, m] \in \textbf{RETURNS} \quad [\mathit{nextpc}(P[pc', m'], pc', m') = pc''] \quad \mathit{objectinvoker} = \mathit{ff} \quad \mathit{callee}(pc', m') = m}{\begin{array}{c} P \vdash \langle pc \cdot pc' \cdot Q, m \cdot m' \cdot M, S'' \odot S' \odot \mathbb{S}, L'' \odot L' \odot \mathbb{L}, F;\ D, C\rangle \\ \longleftrightarrow \langle pc'' \cdot Q, m' \cdot M, S' \odot \mathbb{S}, L' \odot \mathbb{L}, F;\ |S''| \cdot S'' \bullet L'' \bullet D, (\mathit{offset} + pc') \cdot \mathit{end} \cdot pc \cdot C\rangle \end{array}}$$

$$\frac{\begin{array}{c} P[pc, m] \in \textbf{RETURNS} \quad [\mathit{nextpc}(P[pc', m'], pc', m') = pc''] \\ \mathit{objectinvoker} = \mathit{tt} \quad [\mathit{savedobject} = o] \quad \mathit{callee}(pc', m') = m \end{array}}{\begin{array}{c} P \vdash \langle pc \cdot pc' \cdot Q, m \cdot m' \cdot M, S'' \odot S' \odot \mathbb{S}, L'' \odot L' \odot \mathbb{L}, F;\ D, C\rangle \\ \longleftrightarrow \langle pc'' \cdot Q, m' \cdot M, S' \odot \mathbb{S}, L' \odot \mathbb{L}, F;\ o \cdot |S''| \cdot S'' \bullet L'' \bullet D, (\mathit{offset} + pc') \cdot \mathit{end} \cdot pc \cdot C\rangle \end{array}}$$

Exceptions:

$$\frac{[\mathit{trans}(P[pc_1, m_1], pc_1, S_1, L_1) = \mathbf{exn}(pc_{catch}, n, o)]}{\begin{array}{c} P \vdash \langle pc_1 \cdot pc_2 \ldots pc_n \cdot Q, m_1 \cdot m_2 \ldots m_n \cdot M, S_1 \odot \ldots \odot S_n \odot \mathbb{S}, L_1 \odot \ldots \odot L_n \odot \mathbb{L}, F;\ D, C\rangle \\ \longleftrightarrow \langle pc_{catch} \cdot Q, m_n \cdot M, o \cdot \epsilon \odot \mathbb{S}, L_n \odot \mathbb{L}, F;\ (|S_n| \cdot S_n) \bullet L_n \bullet \ldots \bullet (|S_1| \cdot S_1) \bullet L_1 \bullet D, \\ \mathit{startcatch} \cdot \mathit{throwlast} \cdot m_n \cdot pc_n \cdot \mathit{throwcont} \cdot m_{n-1} \cdot pc_{n-1} \ldots \mathit{throwfirst} \cdot m_1 \cdot pc_1 \cdot C\rangle \end{array}}$$

FIGURE 4. Method call and exception operational semantics.

that is, an array of strings and arbitrary values for any other local variables declared in the main method. The initial value of F consists of all the static fields of the program.

We do not represent the effects of a method being marked as `synchronized`, see Section 7.1.

The function *newlocals* constructs the new initial local variables for the method when it is called. The flag *objectinvoker* is true if the method that we are returning from was invoked by one of the three members of **OBJECTINVOKES** and, in the case that it was, *savedobject* is the value of o in the corresponding *INVOKE* instruction. *callee* takes the location of an *INVOKE* instruction and returns the method that it calls. This is needed when reversing a *RETURN*.

In some places method pointers m or m' are placed onto the program counter circular buffer. These are used merely to guide the reversal simulation code.

The operational semantics of method calls and exceptions are given in Figure 4. In Figure 4, the letter m has been used to denote a method, in contrast to Figure 2 where m denoted an integer.

There are four types of method transition which can occur during debugging, namely:

- stepping forwards into the start of a method;
- stepping forwards out of the end of a method;
- stepping backwards into the end of a method;
- stepping backwards out of the start of a method.

The first of these, for example, corresponds to the forwards direction of the two *INVOKE* rules.

The rules given in Figure 2 can be interpreted in the presence of method calls, by looking only at the top stack on the stack of stacks, $\mathbb{S}$, the top local variable frame on the stack of local variable frames, $\mathbb{L}$, and the program counter value on the top of the stack, Q. $P[pc]$ would be replaced by $P[pc, m]$, where m is the method currently on the top of the method stack M and similarly for *nextpc*.

5. ARCHITECTURE

5.1. Overall scheme

We use Kaffe [11] as the host virtual machine for our debugger because it is open-source. We switch off Kaffe's JIT engine because we wish to achieve controlled execution by modifying the interpreter. We implement our debugger for the Linux operating system, but Kaffe runs on many platforms, so in principle it should be straightforward to port.

When debugging, two virtual machines are running. The first is running the user interface. In the case of the command line interface, any JVM can be used; but in the case of the GUI, Swing is required. While editing or compiling code with the GUI this is the only JVM running.

When, however, we start debugging, the user interface creates, as a process, a modified version of the Kaffe JVM running another Java program which we refer to as the client. The user interface creates a server socket and the client creates a client socket, through which all debugging instructions and replies from the debugger pass. Thus, in principle, we could debug across the Internet. The command line interface passes commands typed in directly to the client. Thus, the GUI is entirely built on top of the command line protocol.

The client loads the classes for the target program that we wish to debug, using a ClassLoader. Thus, the client and target are running in the same instance of the virtual machine. The main thread of the client becomes the main thread of the target, but before this happens, the client starts a new thread to listen for instructions from the socket and to act on them.

We need to invoke methods in the debugger, specifically those in the client, from the Kaffe JVM code which is running the target. In fact we can directly call the Java methods in the client from the C code in Kaffe, almost as though the relevant thread in the target had called those methods. Thus, it is easy for the modified Kaffe code to notify the client of matters which need to be reported to the user interface. Essentially, it is as though we had instrumented the bytecode with these method calls. To have done this with our scheme would, however, have resulted in large class files.

Interaction between the client and the target is minimized and only occurs when the GUI must be told of some event so that it can be reported to the user. In particular when running to a breakpoint, the client is only informed of the change in line number when a breakpoint has actually been reached.

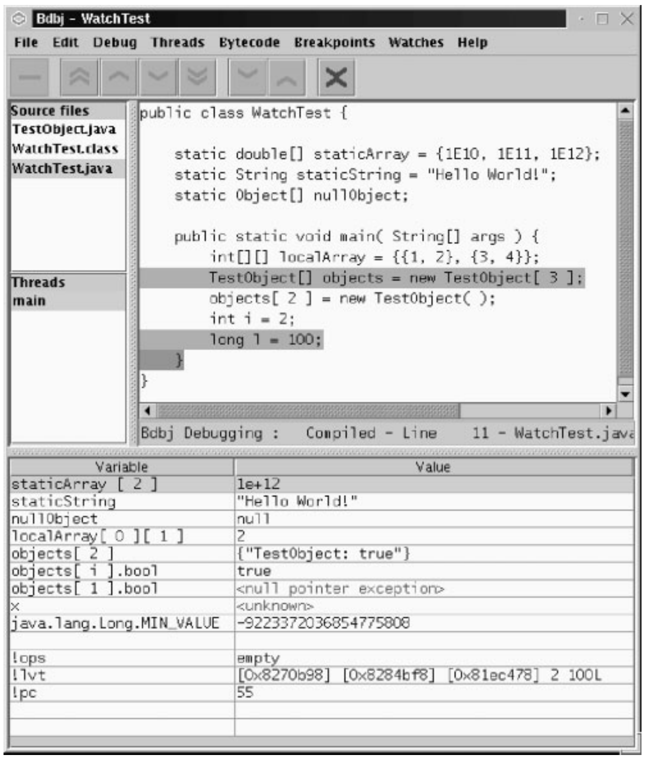
Java was chosen as the language for the GUI so that Swing could be used to construct the user interface. The use of Java here is purely an engineering choice. As we communicate across a socket, the user interface could be written in any language capable of network communication. Java was chosen as the language for the client as its main function is to mediate between a single client socket and a number of threads attempting to read from and write to it. This is a task which can be much more naturally coded in Java than in C.

This scheme is similar to the Java Platform Debugger Architecture (JPDA) [12] which also has a layer which runs in the same virtual machine instance as the target and can communicate with a front-end through a socket. In the case of the JPDA, the communication may be through some channel other than a socket.

We include a screen-shot as Figure 5.

5.2. Single stepping

The modified Kaffe JVM runs normally until it detects that it is running a method contained in one of the source files which the user has specified should be debugged. While in this method, before each bytecode instruction is executed, we check whether we have moved onto a new line of the source code. If so, the modified Kaffe JVM calls the get method of a synchronized first-in-first-out queue in the client. The client adds instructions encoded by integers to this queue as it receives textual commands from the user interface.



```
public class WatchTest {

    static double[] staticArray = {1E10, 1E11, 1E12};
    static String staticString = "Hello World!";
    static Object[] nullObject;

    public static void main( String[] args ) {
        int[][] localArray = {{1, 2}, {3, 4}};
        TestObject[] objects = new TestObject[ 3 ];
        objects[ 2 ] = new TestObject( );
        int i = 2;
        long l = 100;
    }
}
```

```
Bdbj Debugging :   Compiled - Line    11 - WatchTest.java
```

Variable	Value
staticArray [2]	1e+12
staticString	"Hello World!"
nullObject	null
localArray[0][1]	2
objects[2]	{"TestObject: true"}
objects[i].bool	true
objects[1].bool	<null pointer exception>
x	<unknown>
java.lang.Long.MIN_VALUE	-9223372036854775808
!ops	empty
!lvt	[0x8270b98] [0x8284bf8] [0x81ec478] 2 100L
!pc	55

FIGURE 5. Screen-shot.

We use a queue in order to keep the user interface responsive. The user can issue several forward step commands while the debugger is running to a breakpoint and these commands will all be handled when the breakpoint is reached.

When a forwards step instruction is received from the user interface, then, an integer representing STEP is added to the queue in the client. When the modified Kaffe JVM next queries this queue, it receives this integer and runs the target program until the line number next changes.

5.3. Stepping over methods

Stepping over forwards refers to stepping forwards one line without stepping into methods. This is implemented by allowing a special value to be passed to a method call which specifies that the method should run to completion without stopping to wait for instructions.

Stepping over backwards is implemented in an analogous manner.

5.4. Breakpoints

We maintain a table of breakpoints in the modified Kaffe JVM. Each breakpoint consists of a line number and a corresponding source file. We also store a single Boolean value which indicates whether we are currently running to a breakpoint. When the command to run to the next breakpoint is received, this flag is set to true. At the start of each line thereafter, with this flag true, we test whether the new line is a breakpoint. If it is, we clear the flag and block waiting for the next command from the client.

We use a similar scheme to run back to breakpoints: the same breakpoint flag forces the reversal code to run until a breakpoint is reached.

5.5. Which methods are logged?

Normally, logging of each thread begins when the thread first enters a method in a source file which is open in the debugger. Logging ends when that thread enters a method whose source code file is not open and nothing deeper on that thread is logged. The user is able, however, to specify additional classes which should be logged, but not stepped through. These can be used to act as a bridge between two methods which are separated in the call chain by a method in a file which is not open. This feature can also be used to ensure that stateful library class calls are properly reversed. By default all the classes in `java.util` are treated in this way.

Alternatively an 'initial method' can be specified. In this case logging starts with that method and code executed before that method is reached will execute more quickly. This can be particularly useful if the user is only interested in debugging that method, or methods that it calls, and time-consuming code is called before the method is reached.

In addition, it is possible to step through the bytecode rather than the source code. Provided that both the bytecode and source code of a class are open, the user can switch between the two while debugging. Special watch values can be used to inspect local variables and the operand stack. We keep track of the types of the operand stack and local variable slots, so that their contents can be displayed in an appropriate format.

We call all top-level methods (those methods which are not invoked by another Java method) with a special version of the 'forwards into' parameter, which indicates that we should not log the method unless it is in an open file or the initial method. In this case, it passes the normal 'forwards into' parameter to all methods it calls and to all methods they call and so on.

6. GARBAGE COLLECTION

Java employs a garbage collection scheme which deallocates memory when there are no more references to the objects occupying it. We want to ensure that objects are only garbage collected when they can no longer be accessed by either forwards or reverse execution.

We do this by creating a special array of objects in the client. When we push an object to the logging circular buffer, we also add it to this array. The object will then be spotted by the garbage collector and the memory it occupies will not be freed.

It is essential that we do not put values which are not objects, such as integers, into this array as this is likely to crash the garbage collector. Thus, we keep track of which operand stack slots and local variables currently contain objects. Doing this does not have a significant overhead and the logic involved is simple.

We do not need the local variable table to determine which local variables are objects. The first few local variables are the formal arguments to the method and we can deduce which of these are objects by looking at the method's signature. The remainder initially contain trash and so must initially be viewed as not containing objects. Only when they are first initialized to values do we mark them as containing objects or not.

7. MULTI-THREADING

We support multi-threading by a simple mechanism. Each thread of the target program is run by a C thread in Kaffe. For each such thread we create a corresponding synchronized queue in the client. When an instruction is passed to the debugger by the user, the client places the instruction in the relevant queue. When the target thread next queries its queue, it pops off the instruction and executes it.

The user interface has complete control over the direction in which every thread is executing.

Each thread in the program being debugged is allocated its own circular buffers and other necessary flags in the modified Kaffe JVM.

7.1. Locks

The code contained in a synchronized method is protected by a monitor, which can be recursively re-entered. A method which is synchronized in the forwards direction is also synchronized in the reverse direction. *MONITORENTER* and *MONITOREXIT* are treated as mutual inverses. Thus, the critical regions are the same in both directions.

7.2. Determinism

This scheme is not deterministic: subsequent forward runs of the code are subject to different scheduling decisions. However, the user has control over how the threads interact. In particular one thread can be run forwards and another backwards at the same time. It is for this reason that a method running backwards is synchronized if the method would be synchronized running forwards. There cannot be two threads in a critical region at the same time whatever the user does. We rely on the user to take threads back to synchronization points when necessary.

When a thread terminates it can be helpful to hold it in limbo, that is to catch it just before it terminates, so that it is still possible to step back through it. This can be achieved by setting a breakpoint on the return statement. This approach will fail in the case that an exception terminates the thread. In this instance, before the exception is dispatched, the user is asked whether they wish to allow the exception to be dispatched or to step back to the point just before it was thrown.

Work has been done on deterministic *replay* of Java code, see Section 12.3.

TABLE 1. Benchmark results.

	Linpack			jBYTE ($\mathbb{Z}$)			jBYTE ($\mathbb{R}$)			Kopi		
JVM/mode	MFlops	n_1	n_2	Index	n_1	n_2	Index	n_1	n_2	Time	n_1	n_2
Java 1.4.0-rc (mixed)	96.8	1.0		1.77	1.0		1.34	1.0		2.64	1.0	
Kaffe 1.0.6 JIT	25.7	3.8		1.41	1.3		0.819	1.6		1.86	0.70	
Java 1.4.0-rc Interpreter	7.74	13		0.227	7.8		0.214	6.3		2.62	1.0	
Kaffe 1.0.6 Interpreter	2.45	40	1.0	0.0486	36	1.0	0.0678	20	1.0	6.22	2.4	1.0
Debugger: no logging	1.84	53	1.3	0.0406	44	1.2	0.0580	23	1.2	7.27	2.8	1.2
Debugger: logging	0.345	280	7.1	0.0110	160	4.4	0.0162	83	4.2	16.45	6.2	2.6

8. NATIVE CODE AND EXTERNAL RESOURCES

Native methods can be divided into those which perform some mathematical operation and those which access external resources. Mathematical routines do not, in general, need to be reversed as they have no side effects.

It is doubtful whether console I/O should be reversed, as in debugging it can be useful to run the program backwards over a command which reads input from the console, and then give different input.

In Kaffe, file accesses which must be reversed go via the calls: KOPEN, KCLOSE, KREAD, KLSEEK and KWRITE. We intercept all such calls to find whether they occurred on a thread currently in a logged region. If so, we record any necessary data before the real call occurs. KOPEN opens a file and is reversed by closing the file. KCLOSE, when logged, is prevented from closing the file, as on reversal we wish to keep the same file descriptor. The reversal of KCLOSE does nothing. When KREAD or KLSEEK is called, a record is made of the current position in the file. These calls are reversed by seeking back to the recorded position. When KWRITE is called, a note is made on the logging circular buffer of all the data that is to be overwritten. On reversal this data is written back to the file in the correct location.

We have not implemented the reversal of calls which delete files, manipulate the directory structure or involve sockets, graphics or processes.

9. CASE STUDY: REVERSING A COMPILATION

We obtained the source and compiled code of version 1.5B of the Kopi open-source Java compiler [13], which is written in Java. The source code comprises just over 90,000 lines of code including the comments. We started a debugging session, set to log all of the Kopi classes and those of `java.util` and `java.io`.

We then ran the compilation of a simple Java program, which solves the Eight Queens problem, to the end and then back to the beginning using roughly 20 Mbytes of logging space. Having reached the beginning we ran the program to the end again to check that all relevant states had been properly reset by the reversal. We confirmed that the second running of the compilation had generated correct output, that is a class file which solves the Eight Queens problem.

By way of comparison, the Eight Queens program itself can be run fully back to the beginning with about 2 Mbytes of logging memory and Linpack requires roughly 180 Mbytes. This large value for Linpack is a result of its many, often nested, loops.

The ability to reverse the compiler is evidence of both the completeness and the robustness of our debugger. A number of exceptions are thrown during the compilation and they are correctly reversed. At no point is too little or too much pushed to or popped from the buffers.

10. PERFORMANCE MEASUREMENT

We benchmarked our debugger using the Linpack and jBYTEmark benchmarks. For background information on speed performance issues associated with the Java language, see [14]. Table 1 shows the results of the tests, as reported by each benchmark, normalized with a fast dynamic compiler taken to be 1.0 (the n_1 column) and normalized with the unmodified Kaffe interpreter taken to be 1.0 (the n_2 column). All of the Linpack measurements were averages over 10 consecutive runs. The jBYTEmark benchmark consists of several integer tests (the $\mathbb{Z}$ column) and several floating point tests (the $\mathbb{R}$ column). The index value is, in each case, the geometric mean of the results of the tests. The Kopi column is a measurement of the time taken, in seconds, for the Kopi compiler to compile the client part of our debugger.

Thus, logging is roughly 4 to 7 times slower than the Kaffe interpreter, which is in turn roughly 20 to 40 times slower than a recent dynamic compiler, when running the benchmarks on our machine. The measurements were taken on a 1.00 GHz Pentium III with 256 Mbytes of memory, running Linux.

The 'no logging' value refers to executing Java code in methods before the initial method is reached, if that mode is being used. The 'logging' value refers to executing Java code which is actually debugged.

If debugging a long running program and wishing to debug a method which is not called for a long time, the user can set that method to be the initial method. The 'Debugger: no logging' value was obtained by specifying a non-existent method as the initial method when running the benchmark.

There is still a slowdown in this case, because each method must be checked, on first entry, to see whether it is the initial method.

Various optimizations are employed, generally to improve speed at the cost of increased space usage. Information that must be generated which is method specific is generated on the first call to the method and stored for future calls.

Ultimately, however, this technique of reverse execution places an overhead on individual bytecode instructions. This is because some administration is required in the functions which push values to the circular buffers.

This overhead is felt most when running a program which frequently uses low-level bytecode instructions. An instruction like *NEW*, however, takes a relatively long time and the overhead of logging any lost data is small in comparison. Linpack makes use of very few high-level instructions and thus performs worst under our debugger. jBYTEmark uses more high-level instructions and performs better. The effect of logging is far less severe with a program like the Kopi compiler. This is because the code is high-level and involves very expensive operations such as file I/O. Thus, our debugger is most practical for debugging high-level object-oriented code and least practical for debugging numerically intensive code.

Unlike checkpointing techniques, the extra memory usage of our scheme does not depend on the amount of memory normally used by the target.

Tolmach and Appel [6] report timings of their debugger which can reverse execute Standard ML code. They give measurements with the measurement corresponding to an optimizing JIT compiler normalized to 1.0. Their interpreter value is 52, their debugging without logging value is 2.3 and their actual debugging value is 2.7. This suggests that our logging method is 30 to 100 times slower than Tolmach and Appel's checkpointing method. Unlike Tolmach and Appel we do not, however, incur any unusual compile-time costs and we give the user considerable control over how much extra memory is used when debugging. The differences between the source languages SML and Java may impact on the efficiency of various approaches.

11. FUTURE WORK—JIT COMPILER INTEGRATION

It may be possible to JIT compile those methods which are called before the initial method and to only interpret those methods which we wish to log and step through. This would make the tool considerably more practical.

12. RELATED WORK

12.1. Instrumentation

Tolmach and Appel [6] described the use of instrumentation. The abstract states that, 'Traditional source-level debuggers for compiled languages actually operate at machine level, which makes them complex, difficult to port, and intolerant of compiler optimization'.

Despite this, our debugger operates at the virtual machine level, although this is transparent to the user who is able to inspect variables using the names in the source code (see [15]) and to step through the code a source code line at a time. Java's use of a virtual machine with the same bytecode for all platforms removes the porting problem. That is to say, the problems highlighted by Tolmach and Appel do not apply to a virtual machine in the same way that they do apply to native code. Also, Java is compiled with each line of the source code corresponding to a contiguous block of bytecode instructions, which is necessary for the LineNumberTable attribute to be generated. The debugger is tolerant of many of the compiler optimizations used by Java compilers and instructing the compiler to generate full debugging information should ensure that no problematic optimizations occur. Compiled Java class files usually contain line number information.

12.2. Inverse programs

Biswas and Mall [7] developed the idea of the inverses of some statements and hence the inverse of a program. Their approach is to keep a trace file, which stores data on the flow of control and lost states after an assignment and an inverse program which contains the inverses of those statements for which an inverse exists. In the examples given in the paper, relatively few of the statements in the source code have a corresponding statement in the inverse program—generally it is increment, decrement and other arithmetical statements.

Our implementation uses the technique of keeping a record of the flow of control and lost data and for some instructions we implement inverse instructions.

They do not report any benchmarking data on their debugger.

12.3. Multi-threading

Choi and Srinivasan [16] have developed a tool which allows a run of a multi-threaded Java application to be deterministically replayed, which is useful in debugging. The tool records thread scheduling information during the first run of the program, which it uses to reproduce the program's behaviour during a replay.

12.4. TraceBack

TraceBack [17] is a commercial product, which allows Java code to be deterministically replayed, in both directions, after a failure. The technique involves instrumentation of the bytecode file. One advantage of this is that the debugger is not tied to a specific virtual machine. The tool builds a graph of all possible execution paths and uses this to decide where to insert 'agents' in the code which keep track of the flow of control. When a failure occurs, the user can create a 'snapshot' of that execution and then step backwards and forwards within it.

13. CONCLUSIONS

We used logging as our main approach, although the saving of the operand stack and local variables on exit from a method resembles checkpointing.

Checkpointing is characterized by the need to store an amount of data of the order of the memory footprint of the target in order to step back any distance. Certain techniques can be used to reduce this and for a program with small memory requirements this is also acceptable.

With our technique, the distance through which the user can step back is roughly proportional to the size chosen for the circular buffers, regardless of how much memory the target program uses. As we stated above, the user has control over the size of the buffers.

In the situation of a program requiring a large amount of memory, a checkpointing debugger can only store a few checkpoints. In this case, checkpointing involves a tradeoff between the distance that we can step back and the speed of reverse execution, determined by how often we checkpoint. If we checkpoint frequently we can step back quickly, but not very far.

With our technique the speed of reverse execution is constant and we determine how far we can step back by the size of the buffers.

We believe that logging is an elegant technique and that there are many instances where it is less wasteful than checkpointing. The more usual uses of native methods and external resources can be catered for without checkpointing.

We have shown that a simple and intuitive scheme for reversing instructions which only operate on the stack can be extended to the entire JVM language.

Viewed as a debugger for pure Java, we find our tool useful for developing Java applications. (It is also quite enjoyable to experiment with stepping backwards and forwards, particularly with multi-threaded programs.) The reader may wish to obtain our debugger which is available from http://www.dcs.ed.ac.uk/home/jjc/.

ACKNOWLEDGEMENTS

The idea for this project is due to Timothy Harris (University of Cambridge). I would also like to acknowledge the kind assistance of Stephen Gilmore (University of Edinburgh) and of Richard Watts (University of Cambridge) and the financial support of the EPSRC. The helpful comments of the anonymous referees are gratefully acknowledged.

REFERENCES

[1] Sun's JDB homepage. http://java.sun.com/products/jdk/1.1/docs/tooldocs/solaris/jdb.html, 6 May 2002.

[2] Borland's JBuilder homepage. http://www.inprise.com/jbuilder, 6 May 2002.

[3] Sun's Forte for Java homepage. http://www.sun.com/forte/, 6 May 2002.

[4] NetBeans homepage. http://www.netbeans.org, 6 May 2002.

[5] Agrawal, H., DeMillo, R. A. and Spafford, E. H. (1991) An execution-backtracking approach to debugging. *IEEE Software*, **8**, 21–26.

[6] Tolmach, A. and Appel, A. W. (1995) A debugger for Standard ML. *J. Funct. Program.*, **5**, 155–200.

[7] Biswas, B. and Mall, R. (1999) Reverse execution of programs. *ACM SIGPLAN Notices*, **34**, 61–69.

[8] Lindholm, T. and Yellin, F. (1999) *The Java Virtual Machine Specification* (2nd edn). Addison-Wesley Longman.

[9] Stata, R. and Abadi, M. (1998) A type system for Java bytecode subroutines. In *Proc. POPL 98: 25th ACM SIGPLAN–SIGACT Symp. on Principles of Programming Languages*, San Diego, CA, 17–18 January, pp. 149–160. ACM, New York.

[10] Freund, S. N. and Mitchell, J. C. (1999) The type system for object initialization in the Java bytecode language. *ACM Trans. Program. Lang. Syst.*, **21**, 1196–1250.

[11] Kaffe homepage. http://www.kaffe.org/, 6 May 2002.

[12] The Java Platform Debugger Architecture homepage. http://java.sun.com/products/jpda, 6 May 2002.

[13] Kopi open-source Java compiler homepage. http://www.dms.at/kopi/, 6 May 2002.

[14] Kazi, I. H., Chen, H. H., Stanley, B. and Lilja, D. J. (2000) Techniques for obtaining high performance in Java programs. *ACM Comput. Surveys*, **32**, 213–240.

[15] Johnson, J. D. and Kenney, G. W. (1983) Implementation issues for a source level symbolic debugger (extended abstract). *ACM SIGPLAN Notices*, **18**, 149–151.

[16] Choi, J. and Srinivasan, H. (1998) Deterministic replay of Java multithreaded applications. In *Proc. SIGMETRICS Symp. on Parallel and Distributed Tools*, Welches, OR, 3–4 August 1998, pp. 48–59. ACM, New York.

[17] InCert TraceBack homepage. http://www.debugjava.com/, 6 May 2002.