Debugging with Reverse Watchpoint

Kazutaka Maruyama
Dept. of Mechano-Informatics
Grad. School of
Information Science and Technology
The University of Tokyo
kazutaka@acm.org

Minoru Terada
Dept. of Information and
Communication Engineering
The University of Electro-Communications
terada@ice.uec.ac.jp

Abstract

Many programmers have had to deal with an overwritten variable resulting for example from an aliasing problem. The culprit is obviously the last write-access to that memory location before the manifestation of the bug. The usual technique for removing such bugs starts with the debugger by (1) finding the last write and (2) moving the control point of execution back to that time by re-executing the program from the beginning. We call the feature to implement this process "reverse watchpoint" and believe automating it is important.

In this paper we propose a new concept, "position", that is, a point in the program execution trace, as needed for step (2) above. The position enables debuggers to automate the control of program execution to support common debugging activities. We have implemented position in Java with a bytecode transformer. Measurements show that position can be provided with an acceptable amount of overhead.

1 Introduction

The testing and debugging of programs requires so many resources to be thrown into software development. The result of this phase directly affects the quality of products. The ease of debugging is important for efficient development of programs.

Studies about formal system specification with mathematical notation[12], automatic testing by semantics[17], and so on, have continued for long years. These projects, however, are not enough to exterminate all bugs. Programmers must debug programs by hand.

In debugging, we usually use debuggers to inspect the behavior of programs. Debuggers have various functions supported by hardware and operating system[19], but the most functions offered are usually only low-level commands. The most problems programmers want to solve with debugger is in higher-level than the commands provided by debuggers. Those must be manually translated to raw debugging commands by programmers. Programmers have to waste energy thinking about how to use debugger commands while they examine the behavior of programs because of a gap between what debuggers can do and what programmers need. In some cases what programmers want debuggers to achieve is in the same pattern. Automating them is useful for efficient debugging.

In this paper, our purpose is to automate a typical debugging operation which finds the last write-access to a target variable and moves the control point of program execution back to that time by re-executing the program from the beginning. We call the operation *reverse watchpoint*. We will describe the advantage of the automation in later of this paper. We also propose the concept named *position*, which introduces absolute coordinate in program traces and will aid implementing reverse watchpoint by program re-execution. We describe the implementation details for Java programs by bytecode transformation. Overhead measurements of transformed bytecodes are also provided.

Figure 1 shows the structure of our proposal. The rest of this paper is organized as follows. Section 2 describes the problem of existing watchpoint and section 3 proposes "reverse watchpoint" facility which solves the problem of watchpoint. Section 4 proposes "position", "timestamp" and "dynamic breakpoint", and describes how to implement reverse watchpoint using them. Section 5 describes the implementation of these ideas for Java programs by bytecode transformation. Section 6 shows the runtime overhead of transformed bytecodes. Section 7 describes another representation of position without timestamp and compare it to our representation. Section 8 discusses the relevance to other works. Section 9 and 10 describes conclusion and future work respectively.

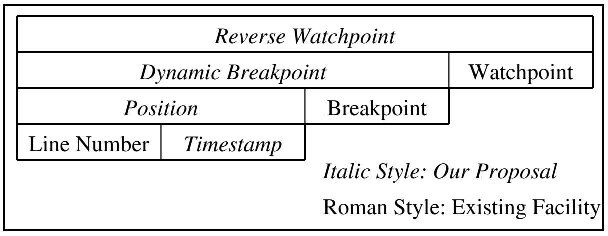

Figure 1. Structure of our proposal.

```
    class List{
      private Cell head;

      public static void main(String[] av){
        List l = new List();
          :
(1)     l.add(c, 1);
          :
(2)     l.print();
      }
        :
      public void add(Cell c, int pos){
        Cell tmp = head;
(3)     for(int i = 0; i < pos; i++){
(4)       tmp = tmp.next;
        }
(5)     c.next = tmp.next;
(6)     tmp.next = c;
      }
        :
    }
```

Figure 2. Code example: `List.java`**.**

2 Problem of watchpoint

It is difficult to fix bugs caused by writing an invalid value to a variable unexpectedly because their manifestation does not occur immediately. *Watchpoint* is suitable for finding such bugs, since it traps all write-accesses to the target variable. When we try to find such kind of bugs, we usually follow the steps below. First, we examine which variable are destroyed. Second, we look for the operation which destroyed it by using watchpoint.

2.1 A sample debuggee

The program being debugged is called *debuggee*. An example of debuggee, shown in figure 2, is a part of the implementation of linked list. The method `add()` adds the given cell to the designated position of a list.

In `main()` method, the caller of `add()`, the cell `c` is

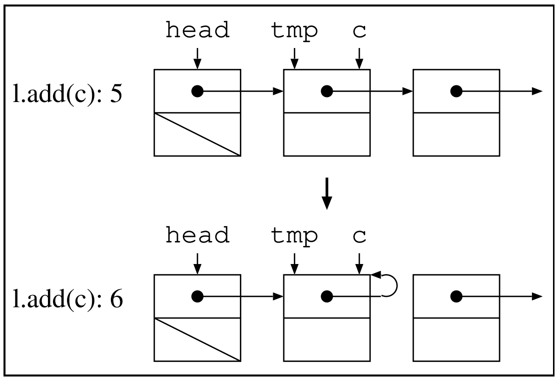

Figure 3. Behavior of `l.add(c)`**.**

inserted into the list `l`. Suppose that `c` erroneously points to the cell which is already a member of `l`. This is what is called *aliasing*. At the line (6) in `add()`, `next` comes to point itself (figure 3), but the program execution still progresses without any errors here. Long after the invocation of `add()` at the line (1), `print()` invoked at the line (2) follows the list from `head` and then the execution falls into an infinite loop. Now, we only know that some errors exist in the list structure.

2.2 Debugging it with watchpoint

To examine the behavior of the debuggee, we set a breakpoint in the method `print()` where infinite loop happens and find a `Cell` object of which `next` field points itself. Since codes of `add()` modify cells, we inspect it. However, we cannot see the cause of the bug there by mere glancing since we are not aware of aliasing.

All we can do is to set a watchpoint at the `next` field of the cell to find out which operation destroyed it. JDB included in JDK-1.4.1 does not support "watchpoint to individual objects" yet, but we describe the following discussion assuming that the support exists.[1] The watchpoint we set gets multiple hits and we will examine the output of the debugger. The output we will get will be like figure 4. Now we can find the true cause of the bug: `tmp` points the cell whose id is 291 and the id of `c` is also 291. They are the same cell.

When we use watchpoint, the debuggee stops many times. We have to examine all the output to find whether the write-access is relevant to the bug. Such work requires very long and careful analysis of the output to find a sign of the bug to be found.

In this simple case, in order to find an object whose `next` points the object itself, all we have to do is to ex-

[1] WatchpointRequest class of JPDA in JDK-1.4.1 now have the capability of adding instance filters by using the `addInstanceFilter` method. JDB is going to have the support soon.

```
> watch c.next
> run List

Field (c.next) is null, will be null:
 "thread=main", Cell.<init>(),
 line=17 bci=11
17        next = null;

  :

main[1] cont

Field (c.next) is slot
 of instance of Cell(id=291),
 will be instance of Cell(id=291):
 "thread=main", List.add(), line=36 bci=28
36        tmp.next = c;
```

Figure 4. Log of JDB session (fictitious).

amine id numbers of objects in the output of the watchpoint. But in general, the condition is not so simple. We usually need to carefully examine the whole output. In any case, the last write to the variable obviously causes the bug. The problem here is that we do not know which write is the last one until the bug manifests. To find that write, we must set a watchpoint at the variable, re-start the debuggee with counting stops at the watchpoint until the control point reaches where bug manifests, and re-start it again to go back to the last write by stopping at the watchpoint with one less counter value. This should be automated. We will describe it in later sections.

3 New debugger command: *reverse watchpoint*

Since there are generally many write-accesses to a variable, it is difficult for programmers to examine the state of the debuggee every time it stops at a watchpoint. It should not be necessary for programmers to go thorough all write-accesses. They are only interested in the last write-access before the bug manifests. We propose the *reverse watchpoint* facility which automatically moves the control point of the debuggee to the last write to a certain variable. This new debugger command takes a variable name to be observed as its argument. The output of the command `revwatch` would be like figure 5.

Using the reverse watchpoint, programmers do not have to care about each stop by watchpoint. They can concentrate their attention on more intelligent work in debugging.

This procedure corresponds to manually creating dynamic slicing[1]. The advantage of our proposal is that control point of execution can be moved to interesting positions wherever a variable is assigned without causing "unconstrained pointers"[4] problems, such as those found in C. Slicing techniques are based on analysis of source codes, but watchpoint of debuggers receive support from hardware for precise information of assignments. Our method is more robust to problems like aliasing because we do not analyze any source codes.

```
> revwatch c.next

Field (c.next) is slot
 of instance of Cell(id=291),
 will be instance of Cell(id=291):
 "thread=main", List.add(), line=36 bci=28
36        tmp.next = c;
```

Figure 5. Log of reverse watchpoint session (fictitious).

4 Implementation of reverse watchpoint

To realize the reverse watchpoint described above, we have to be able to move the control point to the target. So we need to have a way to specify one point in the program trace, which is the series of statements executed in order of time.[2] We call this point *position*. We express it as a pair of a line number and a value of a counter. We use it for the implementation of reverse watchpoint.

4.1 *Timestamp* and *position*

In debugging, the expression of control of program execution used so far is based on static information such as line numbers in source codes. Positions cannot be expressed this way because the backward jumps of the control point can cause multiple executions of one statement. To distinguish them from each other, we introduce a new counter into debuggees. We call the counter *timestamp*, which increases whenever the control point jumps backward.

An example code with a loop structure is shown in figure 6. There are multiple appearances of three lines (1) to (3) in the program trace (figure 7). We call the static point of execution expressed as the line number *location*.

Timestamp increases whenever the loop body is repeated, so the pair of a line number and a value of timestamp expresses a position, the dynamic point of execution. Using

[2] The *point* is really *one statement* of the source code.

```
        :
(1) while(i < a){
(2)   i += b;
(3) }
        :
```

Figure 6. Code with a loop structure.

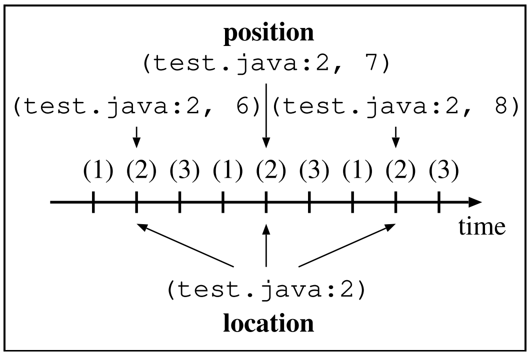

Figure 7. Location and position in program trace.

the value of timestamp, we can distinguish among multiple appearances of a location in the trace.

Position could be expressed as a whole history of debugger commands, rather than the pair. We discuss this topic in section 7.

4.2 Updating timestamp

Timestamp increases when the control point of execution jumps backward. In Java programs, it corresponds to the following cases: (1) entrance and exit of methods, (2) loop bodies and (3) exceptions. All we have to do to increment timestamp is to invoke a static method, `Timestamp.inc()`, just before the execution of the statements above. The current implementation of `Timestamp` class is shown in figure 8.

4.3 Dynamic breakpoint

In this section, we describe how to move the control point to a position. We propose a new breakpoint facility, *dynamic breakpoint*. It will be set at a position rather than a location.[3] Because the implementation of it needs the existing breakpoints, we use Java Platform Debugger Architecture[23] (JPDA) included in JDK. Java debugger JDB also uses JPDA.

[3]We call the normal breakpoint *static breakpoint*.

```
public final class Timestamp{
  private static long ts, ref;

  static{
    ts = 0;
    ref = 0;
  }

  public static void inc(){
    if(++ts == ref) brake();
  }

  private static void brake(){}
}
```

Figure 8. `Timestamp` **class.**

A simple implementation of dynamic breakpoint could use "conditional breakpoint" of debuggers. Although JDB included in JDK-1.4.1 does not provide it, we show an instruction example in JDB with the notation of GDB[20]. GDB supports conditional breakpoint. For example, if we want to stop the debuggee at the position `(test.java:2, 8)`,[4] we instruct JDB as follows:

```
break test.java:2 if Timestamp.ts = 8
```

However, conditional breakpoint evaluates the given condition in the context of the debugger whenever the control point reaches the location shown as △ in figure 9, so it is usually slow. The implementation with the least overhead (only 2 breaks) can be done as follows.

Step 1 Prior to the execution, set a static breakpoint in the method `brake()`, assign the target value of timestamp to the debuggee's variable `ref`. Start the execution until it stops at the breakpoint where the value of timestamp reaches `ref`.

Step 2 Set another static breakpoint at the target location and continue the execution.

4.4 Implementation of reverse watchpoint

Now we describe the implementation of reverse watchpoint. The procedure is as follows.

[4]Debugger already stopped at the position and recorded the timestamp in a previous run. If the second execution path is different from the first, the recorded position becomes meaningless. There is no way the debuggee knows such change by itself. Discussion about non-deterministic debuggees are in section 10.

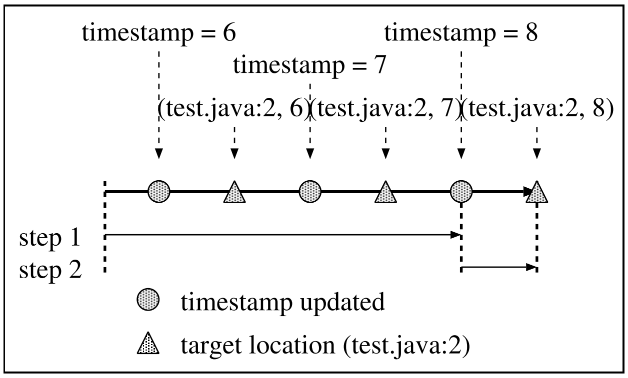

Figure 9. Procedure of dynamic breakpoint.

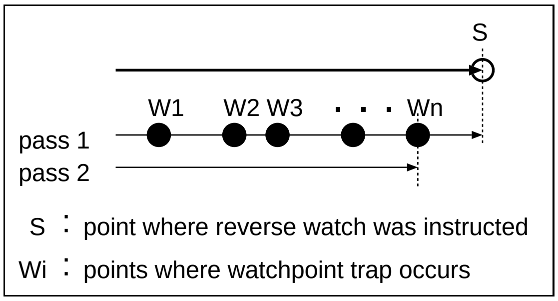

Figure 10. Procedure of reverse watchpoint.

1. Set a dynamic breakpoint at the position where reverse watchpoint is instructed (**S** in figure 10).
2. Pass 1:
   (a) set a normal watchpoint at the target variable and re-start the debuggee. Whenever it stops by the trap of the watchpoint, collect the information for its position (actually the value of the timestamp) in order to mark the position **W1** to **Wn** in figure 10.
   (b) Repeat the collection of positions until the control point reaches the position **S**.[5]
3. Pass 2:
   (a) go back to the beginning again, set a dynamic breakpoint at the most recent position **Wn**, and re-start it.
   (b) The program stops at the target position **Wn**.

5 Timestamp system for java

We need to modify target programs to include codes for updating the timestamp. Prior to Java case, we first describe our experience in C programs.

[5]We can know it by stopping at the dynamic breakpoint set at step 1.

5.1 Implementation for C

We have implemented the transformer in intermediate code level. We considered three levels of C program transformation: (1) source code level, (2) intermediate code level and (3) assembly code level.

The first one has the advantages of independence from hardware platforms, operating systems, and compilers. However, codes programmers see in debugging are different from ones they wrote.

The advantage of the second one is the independence from target architectures if a compiler supports the platform. The disadvantage is dependence on a certain compiler.

The target of the third is assembly code emitted by compilers. We first implemented a simple transformer in this level[16], because it was the easiest method. But it is dependent on target architectures and it lacks portability.

We chose GNU C Compiler (GCC)[21] as the target compiler because it was used in various platforms. GCC generates the intermediate code, called Register Transfer Language (RTL), from the source code and we modified a part of GCC to insert the codes of increment of timestamp at the RTL generation stage.

5.2 Implementation for Java

From our experience of C implementation, we have chosen the implementation using bytecode transformation. We considered four levels to implement the transformer for Java before the decision: (1) compilers (source code level), (2) class files (bytecode level), (3) virtual machines (bytecode execution level) and (4) JPDA (debugger interface level).

The first and the third are excluded because of the same reason as in the case of C. The fourth is very portable but we excluded this approach because of the expected high overhead. Although bytecode can be considered as assembly code in C, the format is established by Java virtual machine specification[15], and it is independent from target architectures.

5.3 Bytecodes which cause increment of timestamp

As explained in section 4.2, timestamp must be updated at several places in the control flow of programs.

entrance and exit of methods No bytecodes correspond to the entrance of methods. The top of method body is used instead. `return` bytecodes (opcode: 172–177) correspond to exit of methods. Bytecodes which invokes methods such as `invokevirtual` could be used for method entrance instead of using the entry of method body. We selected the latter because the amount of added codes is smaller than the former. Another advantage of this choice is that the timestamp overhead is only added to the methods of the modified class. There will be no overhead for calling non-modified classes (such as ones in system library).

branches `ifeq` (153) to `if_acmpne` (166), `ifnull` (198) and `ifnonnull` (199).

gotos `goto` (167) and `goto_w` (200).

other jumps `jsr` (168), `ret` (169) and `jsr_w` (201).

exception `athrow` bytecode throws exceptions, but there exists certain exceptions which are not explicitly `athrow`ed (such as NullPointerException). Because of this, the codes of increment of timestamp are inserted into the entry of the `catch` block instead.

Our bytecode transformer inserts bytecodes for updating timestamp, just before bytecodes described above. The inserted codes are only `invokestatic` followed by the index number of Methodref tag which indicates `Timestamp.inc()` in constant pool. Bytecodes of *branches*, *goto*s and *other jumps* described above have an operand which designates an offset to its target address. The update of timestamp is inserted when the operand has a negative value, i.e. backward jump.

5.4 Bytecode transformer

The transformer which transforms Java class files is written in Java using Bytecode Engineering Library[7] (BCEL) and has around 160 lines.

It is not necessary to transform all class files of a program: it is possible to do only those classes the user considers suspicious. This reduces the overhead significantly.

6 Overhead measurement

We show the results of measurement of runtime overhead and increase of size of transformed class files. The target class files include an empty loop and benchmark programs of SPEC JVM98[22]. We run them under JDK-1.4.0 on Linux PC (Pentium III 733MHz, 640MB memory, and Linux-2.4.7).

We ran each benchmark seven times and took the means of five results except the best and the worst. The result is shown in table 1. In the case of empty loop, whenever the loop body is executed, the method invocation of `Timestamp.inc()` occurs. This benchmark shows the worst case of the overhead of our system. Our implementation at present slows down about 4 times. For other benchmarks overheads are about 1.5 to 2 times. We think it is acceptable for practical use. When the implementation of reverse watchpoint completed, its overhead would be the sum of following two: timestamp system overhead which is about 1.5 to 2 for each of two passes and watchpoint overhead JDB produces. We may estimate the overhead of reverse watchpoint to be less than about 4 times in most cases. Note that `_227_mtrt` benchmark is a multi-threaded program and we added `synchronized` to `Timestamp.inc()` method only for this benchmark. So the overhead is heavier than others.

Table 1. Overhead of Java programs with timestamp system.

Benchmark	Original		Transformed	
empty loop	12.774	(1.00)	56.416	(4.42)
_201_compress	1.4006	(1.00)	2.7754	(1.98)
_202_jess	0.2126	(1.00)	0.271	(1.27)
_209_db	0.412	(1.00)	0.433	(1.05)
_222_mpegaudio	0.1826	(1.00)	0.2792	(1.52)
_227_mtrt	0.363	(1.00)	0.9038	(2.49)
_228_jack	0.603	(1.00)	0.7212	(1.19)

seconds (ratio)

Table 2. Increase of Java class file size.

Benchmark	Original		Transformed	
empty loop	276	(1.00)	328	(1.19)
_201_compress	14443	(1.00)	18640	(1.29)
_202_jess	396536	(1.00)	407240	(1.03)
_209_db	10156	(1.00)	10588	(1.04)
_222_mpegaudio	120182	(1.00)	124438	(1.04)
_227_mtrt	859	(1.00)	920	(1.07)
_228_jack	132516	(1.00)	138109	(1.04)

bytes (ratio)

Table 2 shows the total size of files of each benchmark. The increase is a little large for empty loop benchmark, whose original file size is very small, and `_201_compress` benchmark. But other programs increase only little.

7 Debugger command history as a position

We chose *timestamp representation*, the pair of line number and timestamp, for position representation. Another one is *command-history representation*. When we arrive at a certain position in a debugging session, the whole history of debugger commands enables us to come back to the position by re-executing from the beginning. One of the advantages of our representation is its total order property. Any position can be compared to each other in order of time, so it can be used by applications other than reverse watchpoint.

8 Related work

Boothe[5] made a C debugger with reverse execution capability using a step counter which counts the number of step executions and re-execution from the beginning of debuggees. The capability could be also implemented with our timestamp counter and re-execution. The difference comes from the purpose of each project. Boothe made reverse execution version of existing debugger commands such as "backward step", "backward finish", and so on. For our purpose of providing higher-level program control, the counter granularity of step execution is too expensive.

Feldman et al.[9], Moher[18] and Wilson et al.[25] save complete memory history of process to achieve fully random accessibility to program states. Their systems have to deal with large "log". Our system, however, saves only a pair of line number and value of timestamp to obtain the same capability by assuming the determinism of debuggees.

Lieberman et al.[14, 13] developed a reversible, animated source code stepper, ZStep95. Its modified interpreter saves the order of evaluating S-expression of Lisp programs to provide fully reversible execution. ZStep95 also provides correspondence between a S-expression and a graphical output which is produced by the expression. The correspondence is similar to position, but the interpreter supports only a subset of Lisp and works very slow.

Bertot[2] introduced "occurrences" into the lazy λ-calculus, which makes copies of a subtree in reduction. For example, when an expression e is applied to a lambda function $\lambda x.x + x$, two copies of e will be made and used for both operands of $+$. This can be regarded as creation of multiple positions from one correspondent location at the time of execution. Bertot achieved a breakpoint capability, which is set at an expression and enables the program to stop at any copy of the expression evaluated. In procedural languages, the identification is very easy; use the address of the instruction as a breakpoint. Their purpose is to unify multiple positions to the location; our purpose is to distinguish positions from each other.

Zeller et al.[11, 26] propose "Delta Debugging" which automatically find out data or variables which are concerned with errors by comparing the input data or variables which exit normally with ones which cause errors. It is very useful method and similar with our method in terms of reducing the labor of programmers by power of recent computers. We, however, want to establish more interactive and flexible debugging method and our method is complementary to their one.

Ducassé[8] allows the programmer to control the execution not by source statement orientation, but by event orientation such as assignments, function calls, loops, and so on. Users write Prolog-like forms to designate breakpoints which have complex conditions. This mechanism is complementary to our system and suitable for a front end of it in order to designate appropriate positions where we would move control point to.

Binder et al.[3] integrated a resource management system of CPU and memory into J-SEAL2 mobile agent system using bytecode transformation for complete portability. They use a counter which counts statements executed for CPU resource management and each thread executed updates the counter at each basic block. They reduce the frequency of the update using control flow analysis and the overhead of the system including other components of agent system is 1.41 times of slowing down in the worst case. Hayami et al.[10] also inserted similar codes of counter update using bytecode transformation for the same purpose. They implemented more fine grain management and the overhead is 1.63 times of slowing down. These results are better than ours, because our implementation uses method invocation and its overhead is serious, we think.

9 Conclusion

We proposed a new debugger command, reverse watchpoint, which controls debuggees more abstractly than raw commands of debuggers. In order to implement the reverse watchpoint, we introduced a new idea, position. It marks a point in a program trace using timestamp, which increases at backward jumps of control point of execution.

We also described the implementation details of the timestamp system for Java programs using bytecode transformation under the experience for C programs. The transformer inserts bytecodes of increment of timestamp at certain bytecodes which cause control jumps. Finally we showed the result of runtime overhead and increase of size of transformed class files. We showed that they are acceptable for real use.

10 Future work

The bytecode transformer transforms all the methods in given class files. It would be nicer to have the ability to select methods to reduce overhead.

The driver program of reverse watchpoint for deterministic C programs is completed but one for Java, based on JDB, is still under development. Although JDB included in JDK-1.4.1 does not provide the support of "watchpoint to individual objects", WatchpointRequest class of JPDA in JDK-1.4.1 now have the capability of adding instance filters by using the `addInstanceFilter` method. JDB is going to have the support soon. With this, we can complete the implementation.

For non-deterministic programs, reverse watchpoint do not work well without appropriate replay mechanisms. If non-determinism of debuggees was based on external environment such as input data, we could use some tools to record and replay the environment. For example, Xlab[24] could be used to record and replay X window system events. If debuggees have internal non-determinism such as multi-threading, tools to replay the timing of thread switchings would be needed. Choi et al.[6] implemented a modified Java VM which can replay multi-threaded Java programs. Another way to do replay is by saving the timing via JPDA. We can use one of those techniques to implement reverse watchpoint for non-deterministic programs.

11 Acknowledgments

Thanks to Naoshi Higuchi for the wealth of his knowledge about Java programming and its APIs.

References

[1] H. Agrawal and J. R. Horgan. Dynamic Program Slicing. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 246–256, 1990.

[2] Y. Bertot. Occurences in Debugger Specifications. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 327–337, 1991.

[3] W. Binder, J. G. Hulaas, and A. Villazón. Portable Resource Control in Java. In *Proceedings of OOPSLA'01 the Conference on Object Oriented Programming Systems Languages and Applications*, pages 139–155, 2001.

[4] D. W. Binkley and K. B. Gallagher. Program Slicing. *Advances in Computers*, 43:1–50, 1996.

[5] B. Boothe. Efficient Algorithms for Bidirectional Debugging. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 299–310, 2000.

[6] J.-D. Choi and H. Srinivasan. Deterministic Replay of Java Multithreaded Applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 48–59, 1998.

[7] M. Dahm. Byte Code Engineering. In *Java-Information-Tage 1999(JIT'99)*, 1999.
http://bcel.sourceforge.net/.

[8] M. Ducassé. Coca: A Debugger for C Based on Fine Grained Control Flow and Data Events. In *Rapport de recherche INRIA RR-3489*, 1998.

[9] S. I. Feldman and C. B. Brown. IGOR: A System for Program Debugging via Reversible Execution. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and distributed debugging*, pages 112–123, 1988.

[10] Y. Hayami, K. Taura, and A. Yonezawa. Java Bytecode Transformation for Fine Grain CPU Resource Management. *Transactions on Programming, Information Processing Society of Japan*, 43(SIG3(PRO14)):41–51, 2002. (In Japanese).

[11] R. Hildebrandt and A. Zeller. Simplifying Failure-Inducing Input. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA2000)*, pages 135–145, 2000.

[12] D. C. Ince. *An Introduction to Discrete Mathematics, Formal System Specification, and Z, Second Edition*. Oxford University Press Inc., 1992.

[13] H. Lieberman. Reversible Object-Oriented Interpreters. *First European Conference on Object-Oriented Programming*, 1987.

[14] H. Lieberman and C. Fry. ZStep 95: A Reversible, Animated Source Code Stepper. In B. P. John Stasko, John Domingue and M. Brown, editors, *Software Visualization: Programming as a Multimedia Experience*. MIT Press, 1997.

[15] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification (2nd Edition)*. Addison-Wesley Pub. Co., 1999.

[16] K. Maruyama and M. Terada. One-pass Pseudo Reverse Execution of C Programs. *Transactions on Programming, Information Processing Society of Japan*, 41(SIG9(PRO8)):1–7, 2000. (In Japanese).

[17] B. Meyer. *Introduction to the Theory of Programming Languages*. Prentice Hall, Inc., 1991.

[18] T. G. Moher. PROVIDE: A Process Visualization and Debugging Environment. *IEEE Transactions on Software Engineering*, 14(6):849–857, 1988.

[19] J. B. Rosenberg. *How Debuggers Work*. John Wiley & Sons, Inc., 1997.

[20] R. Stallman, R. Pesch, S. Shebs, et al. *Debugging with GDB – The GNU Source-Level Debugger – Eighth Edition*, 2000. (Included in GDB source tree).

[21] R. M. Stallman. *Using and Porting the GNU Compiler Collection*. Free Software Foundation, 1999. (Included in GCC source tree).

[22] The Standard Performance Evaluation Corporation. *SPEC JVM98 Benchmarks*.
http://www.spec.org/osg/jvm98/.

[23] Sun Microsystems, Inc. *Java Platform Debugger Architecture*.
http://java.sun.com/products/jpda/.

[24] M. Vertes. Xlab – a tool to automate graphical user interfaces. In *Linux Weekly News*, May 1998.
http://mvertes.free.fr/.

[25] P. R. Wilson and T. G. Moher. Demonic Memory for Process Histories. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 330–343, 1989.

[26] A. Zeller. Isolating Cause-Effect Chains from Computer Programs. In *Proceedings of the SIGSOFT 10th International Symposium on the Foundations of Software Engineering (FSE-10)*, pages 1–10, 2002.