

BIT: BYTECODE INSTRUMENTING TOOL

By

HAN BOK LEE

B.S., University of Washington, 1996

A thesis submitted to the
Faculty of the Graduate School of the
University of Colorado in partial fulfillment
of the requirement for the degree of
Master of Science
Department of Computer Science

1997

This thesis for the Master of Science degree by

Han Bok Lee

has been approved for the

Department of

Computer Science

by

Benjamin Zorn

Dirk Grunwald

Vincent Heuring

Date _____

Lee, Han Bok (M.S., Computer Science)

BIT: Bytecode Instrumenting Tool

Thesis directed by Associate Professor Benjamin Zorn

BIT (Bytecode Instrumenting Tool) is a set of interfaces that allows one to build customized tools to instrument Java Virtual Machine (JVM) bytecodes. To better understand program behavior, researchers and software developers have built numerous tools that carry out program analysis. Understanding the dynamic behavior of a program is important in improving program performance, in identifying critical parts of a program, and in measuring the effectiveness of different algorithms.

The JVM is an abstract machine specification designed to support the Java programming language, which has become very popular during the last few years. Although there are tools that analyze and modify executables on a variety of operating systems and machine architectures, there currently is no framework for carrying out the same task for bytecodes, which are the binaries that the JVM interprets. In this thesis, we describe BIT, which allows researchers and software developers to instrument bytecodes to understand their dynamic behavior.

To design and implement BIT, we investigated and studied different approaches for instrumenting executables as well as the inner workings of the JVM and the `class` files that contain bytecodes. We implemented BIT using the Java programming language, and it can be used on any platform that has an implementation of the JVM. BIT allows the user to insert calls to analysis methods anywhere in the bytecode, so that information can be extracted from the user program

while it is being executed. This information, in turn, could be used in performance measurement and optimization.

BIT is the first framework that allows researchers and software developers to create customized tools to analyze JVM bytecodes. A number of different tools ranging from dynamic instruction counters to path profilers could be built using BIT. In this thesis, we describe several customized applications that use BIT and also report on BIT's performance. We found that for a smaller program, the overhead for the execution speed and size were between 15% to 37%, while for a larger program, the overhead for the execution speed was about two times the original execution time.

ACKNOWLEDGEMENTS

First, I would like to thank everyone who helped me with his or her suggestions and support during my master's study at the University of Colorado at Boulder. I would like to especially thank my thesis advisor, Benjamin Zorn, whose valuable insights, helpful suggestions, and patience were central in completing this thesis. I would also like to extend my thanks to the members of my master's thesis committee, Dirk Grunwald and Vincent Huring, for their helpful suggestions.

I would like to express my sincere thanks to my parents, Chun-Koo and Kyung-Ja Lee, whose love and confidence in me made this thesis possible.

Last, but certainly not least, I would like to thank Jee-Young Lee for her moral support during my master's study as well as for her encouragement and love that made it possible for me to reach this point in my life.

CONTENT

CHAPTER

I.	INTRODUCTION.....	1
II.	RELATED WORK	6
III.	BIT SYSTEM ARCHITECTURE	11
IV.	BIT INTERFACE	16
	CLASSES.....	16
	BUILDING AND NAVIGATING OBJECTS.....	17
	CLASSINFO	17
	ROUTINE	18
	BASICBLOCK.....	19
	INSTRUCTION	19
	GETTING INFORMATION.....	20
	CLASSINFO	20
	ROUTINE	20
	BASICBLOCK.....	21
	INSTRUCTION	21
	INSERTING METHOD CALLS	22
	ROUTINE	22
	BASICBLOCK.....	22
	INSTRUCTION	23

V.	BIT IMPLEMENTATION.....	26
	RATIONALE.....	26
	INTERMEDIATE REPRESENTATION.....	27
	ADDING METHOD CALLS.....	28
VI.	EXAMPLE APPLICATIONS.....	30
	PERFORMANCE.....	30
	IMPROVING PERFORMANCE.....	33
VII.	SUMMARY.....	35
	REFERENCES.....	38
	APPENDIX	
	A. ICount.class.....	40

TABLES

Tables

1. Time Required to Instrument User Programs.....	31
2. Execution Time of Instrument User Programs.....	31
3. Code Size of Instrument User Programs	33

FIGURES

Figures

1. Instrumented Program-Based Tracing.....	2
2. Inner Workings of BIT	11
3. Instrumentation Code: Branch Counting Tool	13
4. Analysis Code: Branch Counting Tool.....	15
5. A Sample Java Program that Accesses a Member Variable.....	23
6. A Compiled Java Program that Accesses a Member Variable.....	24

CHAPTER I

INTRODUCTION

It is sometimes beneficial to be able to measure and understand both the static structure and dynamic behavior of a program. Such information could be used in identifying critical pieces of code, predicting the performance of several important algorithms (e.g., branch prediction, cache replacement, instruction scheduling), debugging, and in allowing profile-driven optimizations (Srivastava and Eustace 1994). Furthermore, this information could help in the design and tuning of both hardware and software systems (see Cmelik and Keppel 1994). Over the years, researchers have built numerous tools that allow them to obtain this information.

There are several methods of measuring and understanding the dynamic behavior of a program (Stunkel, Janssens, and Fuchs 1991). One of these methods is hardware-based tracing where hardware directly monitors the program's run-time behavior (Clark 1983). Although its accuracy tends to be high, it is costly and inflexible since the hardware has to be modified each time different information needs to be obtained. Also, hardware-based tracing captures all the events from the operating system, run-time system, and application program. Although this feature can be advantageous in some situations, it can also be disadvantageous because it can be difficult to distinguish the event's origin from the trace. Another method of measuring dynamic behavior of a program is to use simulation-based techniques where the simulation models much of the real hardware, operating system, and run-time system (So et al. 1988). This method provides the potential for highly accurate

traces depending on how complete the simulation is, but its speed is slow. We can also obtain the dynamic behavior of a program by using instrumented program-based techniques in which the user program is directly modified. Figure 1 shows one possible technique employing this method based on ATOM (Sristava and Eustace 1994).

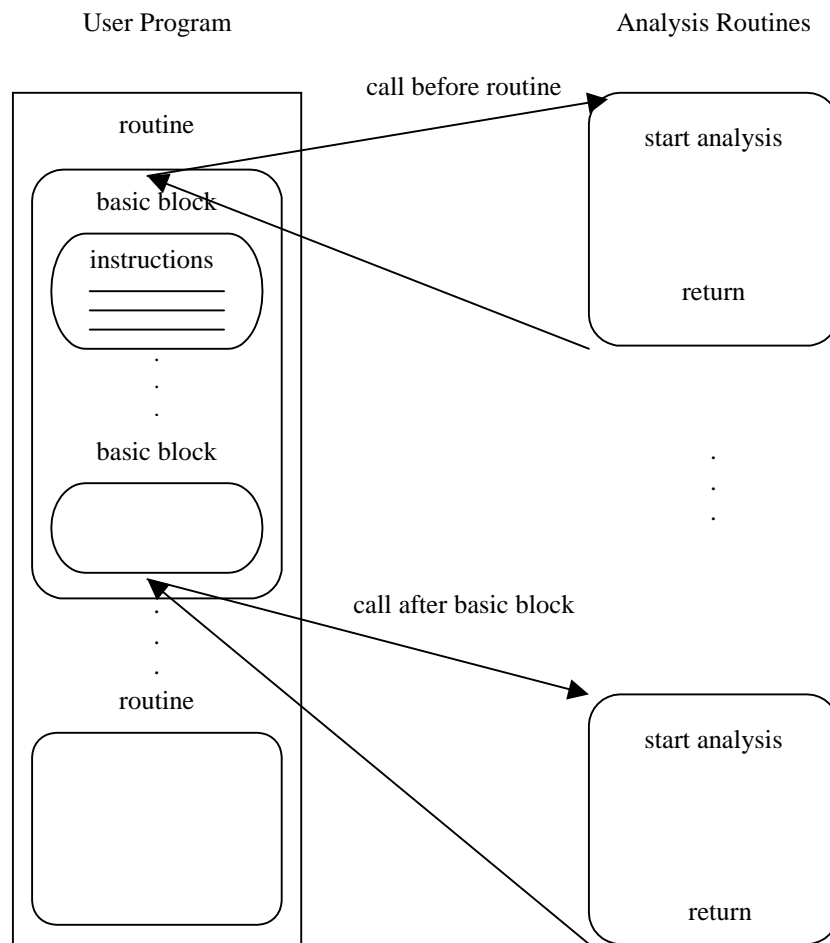


Figure 1. Instrumented Program-Based Tracing

This figure shows a routine as a collection of basic blocks, which, in turn, are viewed as a collection of instructions. A user program may have any number of routines, and calls to analysis methods can be inserted anywhere in the program to obtain run-time information. In this figure, arrows represent procedure calls and returns. These techniques have received attention recently since they are more flexible than hardware-based techniques and yet do not require the complexity of simulation-based techniques (Stunkel, Janssens, and Fuchs 1991).

This thesis describes BIT, which employs an instrumented program-based technique similar to the one shown in Figure 1 for extracting dynamic behavior of Java Virtual Machine (JVM) bytecodes. JVM is an abstract machine specification designed to support the Java programming language, and JVM bytecodes are equivalent to binaries on other machines (Lindholm and Yellin 1997). Although there are tools that allow binary instrumentation on a number of different operating systems and machine architectures, there currently is no framework that allows bytecode instrumentation. BIT is a new set of library classes that allow the user to observe the dynamic behavior of programs by inserting calls to user analysis methods in bytecodes. BIT not only works for Java programs but it also works for any other programming language that can be compiled or translated to JVM bytecodes since BIT operates on bytecodes, and it requires no source code. We implemented BIT using the Java programming language, and therefore, it can be used on any platform that has an implementation of the JVM. Different program analysis tools for measuring and understanding the dynamic behavior of a program can be easily built by using the set of classes that this thesis describes. BIT consists of 42 classes and

about 5,200 lines of code including comments. Current users of BIT include Todd A. Proebsting, who is using it to evaluate the performance of Java bytecodes generated from Icon (Proebsting 1997) source code, Bradley Calder, who is considering using it to build a customized tool to learn better ways of laying out Java bytecode instructions (Hashemi, Kaeli, and Calder 1997), and other researchers at the University of Colorado at Boulder, who are building a tool for tracing the execution of a Java persistent store. Since the Java programming language (Gosling et al. 1996) and JVM bytecodes (Lindholm and Yellin 1997) are relatively new and the performance behavior of JVM (Java Virtual Machine) on a variety of programs is not yet fully understood, many interesting tools can be built using BIT to explore this area.

In addition, this thesis describes the design and implementation of BIT. A sample application that uses BIT is also presented, and we report on BIT's performance. We found that for a smaller program, the overhead for the execution speed and code size was between 15% to 37%, while for a larger program, the overhead for the execution speed was about two times the original execution time.

This thesis is organized to present the reader with a general understanding of binary instrumentation and then shows how BIT accomplishes instrumentation of bytecodes. In Chapter II, we take a closer look at existing tools that carry out binary instrumentation and classify them according to their design goals. We also tell you which tools influenced the design of BIT. The system architecture of BIT is discussed in Chapter III with a sample program that uses BIT. This chapter outlines the process of instrumenting bytecodes. In Chapter IV, four classes that constitute

BIT are presented with their methods. These methods are classified and presented according to their functional categories. We present the details of the implementation in Chapter V. In this chapter, we describe how we add method calls in the bytecodes and also how we organized the classes in BIT. Example applications that use BIT are presented in Chapter VI along with their performance measurements.

Finally, in Chapter VII, we conclude with a summary of the major issues addressed in the thesis. We also present issues that need to be addressed in the near future and how we might go about addressing them.

CHAPTER II

RELATED WORK

This chapter presents many existing tools that employ instrumented program-based techniques to obtain dynamic behavior of programs. First, customized tools that carry out a specific task are presented, and their limitations are outlined. More flexible tools that provide frameworks for building customized tools are also discussed, and we show how their design influenced BIT. Other tools that carry out post-processing on `class` files are reviewed including their limitations. Finally, the differences between BIT and other existing tools are presented.

As it was mentioned in the introduction, there are many tools that employ instrumented program-based techniques to carry out many different tasks. These tasks range from emulation and tracing to optimization (Larus and Schnarr 1995). The Wisconsin Wind Tunnel architecture simulator (Reinhardt et al. 1993), for example, allows the emulation of a cycle counter, which the underlying hardware does not provide. Instrumented program-based techniques have also been used in optimization such as in (Srivastava and Wall 1994) and (Srivastava and Wall 1993). However, most of the tools that have been developed using instrumented program-based techniques are used for studying program or system behavior. Tools such as QPT (Larus and Ball 1994), Pixie (Smith 1991), and Epoxie (Wall 1992) count the number of times each basic block in a program is executed. MPTRACE (Eggers et al. 1990) and ATUM (Agarwal et al. 1986) generate data and instruction traces, and PROTEUS (Brewer et al. 1991) and Shade (Cmelik and Keppel 1994) emulate other

architectures. Also, software testing and quality assurance tools that detect memory leaks and access errors such as Purify (Hastings and Joyce 1992) catch programming errors by using these techniques. Purify inserts instructions directly into the object code produced by existing compilers. These instructions, in turn, check the validity of every memory access performed by the program and report when there are errors.

There are limitations to these tools, however, since they are designed for a specific task and are difficult if not impossible to modify to meet users' changing needs. It would be difficult, for example, for a user to modify a customized tool to obtain more or less detailed information about a trace than what is already provided. To modify a customized tool, a user has to have access to the source code and have a good understanding on how the tool works, including low-level details that deal directly with modifying the binaries. Moreover, many of these tools use inter-process communication or files to relay program behavior to the analysis routines, which are expensive (Srivastava and Eustace 1994).

There is another group of tools, which have different design goals. The tools in this group include OM system (Srivastava and Wall 1993), EEL (Larus and Schnarr 1995), ATOM (Srivastava and Eustace 1994), and Etch (Bershad et al. 1997). These tools differ in that they offer a library of routines for modifying executable files. Users, in turn, can design and build their own customized tools to meet their needs using these tools.

OM works on object files, and it represents machine instructions as RTL, which can later be manipulated and written back to the disk in the form of machine instructions. EEL also uses an intermediate representation to represent machine

instructions. The difference between OM and EEL is that while OM uses relocation information in the object files to relocate edited code, EEL analyzes and modifies the program's instructions directly. Furthermore, EEL can edit fully-linked executables and emphasizes portability in its design. However, EEL currently works only on workstations with SPARC processors, under Solaris and SunOS, and therefore its claim on portability is yet to be realized.

ATOM provides a framework on top of OM, and a number of customized tools from basic block counting to cache simulators can be built on top of that framework. ATOM, unlike OM and EEL, does not allow one to arbitrarily modify the object code, but simplifies the instrumentation process by providing an API to access high-level constructs such as procedures, basic blocks, and instructions, and it also provides a library to easily manipulate those constructs. These library routines include operations such as iterating through these high-level constructs and inserting procedure calls before and after these constructs. This comes at a price, however, since ATOM does not allow removing or replacing existing instructions in the binary files as EEL does. Another drawback of ATOM is that it is not portable. Currently, ATOM runs only on the Alpha AXP under OSF/1.

Etch (Bershad et al. 1997) is an application program performance evaluation and optimization system running on Intel x86 platforms running the Windows/NT operating system. Etch allows the user to instrument existing binaries with arbitrary instructions.

The tools mentioned above operate on object codes for a variety of operating systems and architectures, but none of them work on JVM class files. However, the

Java interpreter provides some profiling information, which includes the method invocation sequence and the size of objects allocated, when invoked with *prof* switch. There also are tools that carry out post-processing on `class` files such as `osjcfp` (ODI 1997) and the work by Cattell (Moss and Hosking 1996) to make classes persistence-capable. However, the inner workings of these tools have not been published. Furthermore, these are also customized tools and have the same limitations mentioned above.

BIT overcomes these limitations by providing a set of classes that users can employ to build their own program analysis tools. BIT allows a user to observe the dynamic behavior of programs by inserting calls to analysis methods anywhere in the bytecode.

A number of factors influenced the design and implementation of BIT including but not limited to those mentioned above. We followed EEL's object oriented design, but also opted for the simplicity that ATOM provides to both the designers of the system and the users.

There are significant differences between the JVM and the other platforms on which ATOM, EEL, and Etch run. First of all, the JVM is a stack-based rather than register-based machine, and therefore, there are no registers to save and deal with other than the program counter and stack pointer. The JVM uses a very uniform set of instructions (all but two bytecode instructions are fixed length), and there are no delayed branches or complex jump tables that have to be dealt with. We believe that these two features alone make the JVM a much easier architecture to work on for building instrumentation libraries.

It is also possible to obtain dynamic behavior of bytecode programs by modifying the JVM, but this requires access to JVM source and can also become very complex to modify to meet changing users' needs.

The Java programming language is both portable and architecture neutral (Gosling et al. 1996), and its class file format is the same for all platforms (Lindholm and Yellin 1997). Therefore, BIT, which is written entirely in Java, is also portable across all platforms. Moreover, the Java programming language is becoming increasingly popular, and tools such as BIT could prove to be very valuable in a number of different areas.

CHAPTER III

BIT SYSTEM ARCHITECTURE

This chapter presents the architecture of BIT including the overall process of bytecode instrumentation and how BIT works internally. An example program is provided to guide the reader through this process.

Like ATOM, the architecture of BIT is based on the observation that many of the dynamic behaviors of a program can be obtained by instrumenting a few key locations, e.g., before and after methods, before and after basic blocks, and before and after instructions. Thus, BIT provides classes and methods for inserting a method invocation at each of these key locations. BIT uses an internal representation to represent bytecodes, to which modification can be made and then written back to a class file. Figure 2 shows how BIT works internally.

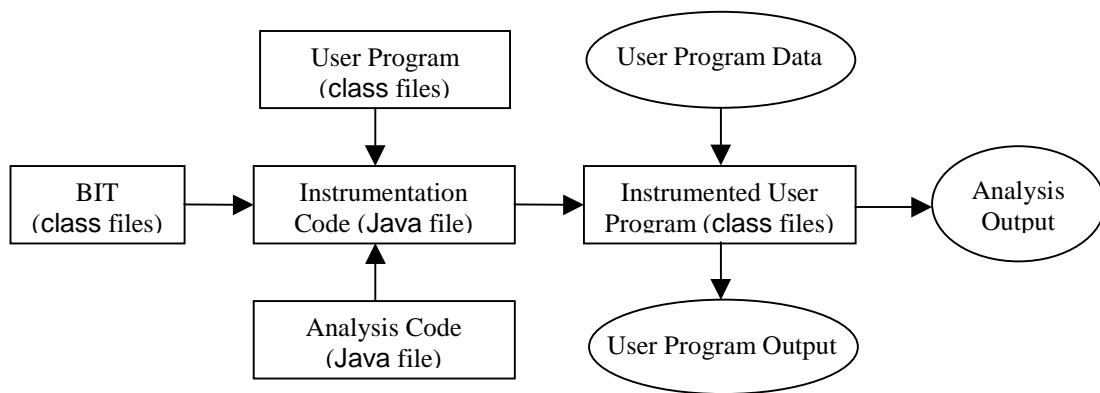


Figure 2. Inner Workings of BIT

The user writes his or her own instrumentation by using the classes and methods that BIT provides. She or he also needs to write analysis code. When the JVM runs instrumentation code, it will insert analysis code at appropriate places in the user program. This process results in the instrumented user program, which then can be executed under the JVM to produce the analysis output.

In this paper, a customized tool that could aid in branch prediction is presented as in (Srivastava and Eustace, 1994). This tool counts the number of branches taken and not taken at all the branches in different methods. Figure 3 shows the instrumentation code for this tool. BIT's methods are shown in bold. This instrumentation code specifies where the user program is to be instrumented and what methods are to be invoked. This tool takes two arguments: an input file and an output file. The input class file is opened and broken down into more manageable pieces (class, routine, basic block, and instruction), which are then instrumented. In this instrumentation program, we first analyze the input file by creating a new `ClassInfo` object, whose constructor parses the input file and stores the intermediate representation in its members. A call to the **`getRoutines()`** method is invoked to obtain the vector of `Routines`. A `Routine` represents a method in the input class file. For each of these `Routines`, we obtain the basic blocks by invoking the **`getBasicBlocks()`** method. To count all the conditional branches in a program, we need to insert analysis code wherever there exists a conditional instruction in the program. This is accomplished by looking at each basic block and seeing whether the last instruction in that basic block is a conditional instruction or not.

```

import classFileVisit.*;

import java.io.*;
import java.util.*;
public class BranchPrediction {
    static DataOutputStream data_out = null;
    static Hashtable branch = null;
    static int pc = 0;

    public static void main(String argv[]) {
        String infilename = new String(argv[0]);
        String outfilename = new String(argv[1]);
        ClassInfo ci = new ClassInfo(infilename);
        Vector routines = ci.getRoutines();
        // for all routines
        for (Enumeration e = routines.elements(); e.hasMoreElements(); ) {
            // this is how one gets an element from an enumeration
            Routine routine = (Routine) e.nextElement();
            // these two methods need be invoked because we need to deal
            // with both instructions and basic blocks
            routine.analyzeCode();
            routine.findBasicBlocks();
            Vector instructions = routine.getInstructions();

            // for all basic blocks
            for (Enumeration b = routine.getBasicBlocks().elements();
                b.hasMoreElements(); ) {
                // get a basic block
                BasicBlock bb = (BasicBlock) b.nextElement();
                // find the instruction at the end of the basic block
                // first get the ending address of this basic block, and use it to
                // get the instruction
                Instruction instr =
                    (Instruction)instructions.elementAt(bb.getEndAddress());
                // look up this instruction's type
                short instr_type =
                    InstructionTable.InstructionTypeTable[instr.getOpcode()];

                // only instrument conditional instructions
                if (instr_type == InstructionTable.CONDITIONAL_INSTRUCTION) {
                    instr.addBefore("BranchPrediction", "Offset", new
                        Integer(instr.getOrigOffset());
                    instr.addBefore("BranchPrediction", "Branch", new
                        String("BranchOutcome"));
                }
            }
            // find out the name of this method
            String method = new String(routine.getMethod());
            routine.addBefore("BranchPrediction", "EnterMethod", method);
            routine.addAfter("BranchPrediction", "LeaveMethod", method);
        }
        // we need to write back the instrumented user program
        ci.write(outfilename);
    }
}

```

Figure 3. Instrumentation Code: Branch Counting Tool

This approach is faster than going through all the instructions and seeing which ones are conditional instructions since there are fewer basic blocks than there are instructions. Once we find conditional instructions, all we have to do is to insert calls to analysis methods before these conditional instructions are executed. In addition, calls are made when entering and leaving a method so that variables are initialized and results are printed.

Note that for each Routine, calls to the **analyzeCode()** method and to the **findBasicBlocks()** method were invoked. These two methods are necessary to break down the code buffer of a class file, which contains bytecode instructions, into BIT's internal representation of instructions and to find the basic blocks. These methods could have been part of Routine's constructor, but we chose not to automatically call them since some instrumentation code might not need access to them and would have slowed them down unnecessarily.

Figure 4 shows the analysis code that is invoked when conditional instructions are encountered. The **LeaveMethod()** method prints the statistics gathered during this method's execution. The **Offset()** method puts the offset of the conditional instruction being executed into a static variable named **pc**, and the **Branch()** method increments branch outcome counters. Notice that the analysis code is defined in the same class as the instrumentation code. Although the analysis code can be defined in a separate class, there is no need to do so. The analysis code uses class variables **branch** and **pc** because BIT does not allow passing more than one argument to the analysis methods at this time. This is a shortcoming that will soon be fixed.

CHAPTER IV

BIT INTERFACE

This chapter presents the design of BIT's interface including the classes and methods that BIT provides according to their functionality. First, a general description of the classes is presented followed by the description of the methods that allow one to build and navigate through objects. Next, the methods that allow one to gather information about the class file being instrumented are presented followed by the description of the methods that allow one to insert calls to analysis methods.

CLASSES

BIT consists of 43 Java classes. In this section, we discuss four major Java classes that analyze and modify a class file: **ClassInfo**, **Routine**, **BasicBlock**, and **Instruction**. A **ClassInfo** object captures the structure of the class file to be analyzed and instrumented. An analysis tool opens a class file by using **ClassInfo** and writes it back to a file by using its **write()** method. A **ClassInfo** contains a vector of **Routines**, in which the actual bytecode instructions as well as intermediate representations are stored. A **Routine** corresponds directly to a method defined within a class file. A **Routine** contains a vector of type **BasicBlock** and a vector of type **Instruction**. A **BasicBlock** contains the start and ending addresses of a basic block found within a method while an **Instruction** represents a Java bytecode instruction. Any of these abstractions can be manipulated individually through the methods that it provides.

It is important to note that when a `class` file is instrumented, its bytecode offsets remain unchanged. Therefore, when instructions are added before and after instructions as in Figure 2, the offsets of conditional instructions that get printed are the actual offsets and not the offsets of the instrumented program. BIT accomplishes this by keeping a separate record of an instruction's original offset in the `Instruction` class.

We decided to provide these abstractions because they contain the boundaries and key locations in which instrumentation code can be inserted for maximum effectiveness. This follows ATOM's design.

BUILDING AND NAVIGATING OBJECTS

This section describes how to build and navigate through objects of different classes. BIT uses Java's `Vector` class to store a collection of objects such as routines, basic blocks, and instructions. The `Vector` class is a resizable array in which Java objects can be stored and retrieved. It is also useful to know about the `Enumeration` class in Java, which is essentially an iterator class. The `Enumeration` class provides two methods: the `hasMoreElements()` method that returns true if there are more elements to iterate over, and the `nextElement()` method that returns the next object.

CLASSINFO

- `public ClassInfo(String filename)` parses and maintains class structure given a name of a class file. More specifically, the constant pool is read and parsed, the methods are parsed and put into a new vector, attributes about this class and its super class are parsed and stored, and finally, different fields and other attributes are also parsed and maintained

internally. As shown in the above example, this constructor can be used as follows: `ClassInfo ci = new ClassInfo(infilename);`

- `public Vector getRoutines()` returns the vector containing *Routine* objects.

The following fragment of code illustrates how to navigate through this vector:

```
for (Enumeration e = ci.getRoutines().elements();
     e.hasMoreElements(); ) {
    Routine routine = (Routine) e.nextElement();
    ...
}
```

ROUTINE

- `public Routine(Method Info method, Cp Info[] constant pool, ClassInfo classinfo)` creates a new *Routine* object that contains a bytecode buffer and other information about this method. Note, however, that instructions and basic blocks are not yet parsed since it might not be needed in the analysis code. If one wants to navigate through basic blocks and objects within this method, one or more of the following methods have to be also invoked. The purpose of separating these processes is to gain some speed during class file processing. Users need not invoke this constructor directly as it is done by *ClassInfo* when its constructor is called.
- `public void analyzeCode()` analyzes the code buffer and breaks it down into a vector of instructions. This method needs to be invoked if one wishes to instrument a bytecode instruction.
- `public void findBasicBlocks()` constructs a vector containing basic blocks in this method. This method needs to be invoked if one wishes to

instrument a basic block. This method can only be invoked after the `analyzeCode()` method's invocation.

- `public Vector getInstructions()` returns the vector that contains the instructions in this method. This method can be used in the following manner:

```
for (Enumeration e = routine.getInstructions().elements();
     e.hasMoreElements(); ) {
    Instruction instruction = (Instruction) e.nextElement();
    ...
}
```

- `public Vector getBasicBlocks()` returns the vector that contains the basic blocks in this method. This method can be used in the following manner:

```
for (Enumeration e = routine.getBasicBlocks().elements();
     e.hasMoreElements(); ) {
    BasicBlock basicblock = (BasicBlock) e.nextElement();
    ...
}
```

BASICBLOCK

- `public BasicBlock(Routine r, int start)` creates a new basic block starting at start address. This constructor is invoked by the `findBasicBlocks()` method in a routine. There is no need to invoke this constructor directly.

INSTRUCTION

- `public Instruction(int opcode, int offset, Routine routine)` creates a new instruction with corresponding opcode. This constructor is invoked by `analyzeInstructions()` in a routine. There is no need to invoke this constructor directly.

GETTING INFORMATION

These methods provide information about the class file being instrumented.

CLASSINFO

- public String getClass() returns the name of the class file being instrumented.
- public String getSuperClass() returns the name of the super class of the class file being instrumented.
- public String getSourceFileName() returns the name of the source file of the class file being instrumented.
- public int getRoutineCount() returns the number of methods present in the class file being instrumented.

ROUTINE

- public short getAccessFlags() returns this method's access flags.
- public int getBasicBlockCount() returns the number of basic blocks in this method.
- public int getCodeLength() returns the length of the actual code buffer in bytes.
- public String getDescriptor() returns this method's descriptor. A descriptor is a string representing the type of a method (Lindhold and Yellin 1997).
- public int getInstructionCount() returns the number of instructions in this method.

- public short getMaxLocals() returns the maximum number of locals of this method.
- public short getMaxStack() returns the maximum stack size of this method.
- public String getMethod() returns the name of this method.

BASICBLOCK

- public int getStartAddress() returns the starting address (in number of instructions) of this basic block.
- public int getEndAddress() returns the ending address (in number of instructions) of this basic block.

INSTRUCTION

- public short getInstructionType() returns the type of this instruction. The types of instructions include nop, constant, load, store, stack, arithmetic, logical, conversion, comparison, conditional, unconditional, class, object, exception, instruction checking, monitor, and other instructions. The classification of instructions into these categories was partially based on the JVM's classification and is described in the `InstructionTable` class in `BIT`'s package.
- public int getOpcode() returns the opcode of this instruction. This opcode can then be used as an index to `OpcodeName` table defined in the `InstructionTable` class to obtain a `String` representation of this instruction.

- `public int getOffset()` returns the original (uninstrumented) offset of this instruction.

INSERTING METHOD CALLS

Being able to insert method calls at various points of interest in the bytecode is the essential requirement for this kind of tool. BIT allows a user to insert method calls before and after each routine, basic block, and instruction. Currently, only one argument of either `String` or `Integer` class can be passed to the analysis routine. There is no inherent characteristic of BIT that prevents passing more than one argument to analysis routines, and this limitation will be removed in the future.

For all of the methods below, a run-time check is done to make sure `arg` is an instance of either the `String` or `Integer` class. An error is raised if this is not the case.

ROUTINE

- `public void addAfter (String cname, String mname, Object arg)` adds a call to method *mname* defined in class *cname* after this routine.
- `public void addBefore(String cname, String mname, Object arg)` adds a call to method *mname* defined in class *cname* before this routine.

BASICBLOCK

- `public void addAfter(String cname, String mname, Object arg)` adds a call to method *mname* defined in class *cname* after this basic block.
- `public void addBefore(String cname, String mname, Object arg)` adds a call to method *mname* defined in class *cname* before this basic block.

INSTRUCTION

- `public void addAfter(String cname, String mname, Object arg)` adds a call to method *mname* defined in class *cname* after this instruction.
- `public void addBefore(String cname, String mname, Object arg)` adds a call to method *mname* defined in class *cname* before this instruction.
- If *arg* is equal to a String object whose value is “BranchOutcome,” then branch condition is passed to the analysis method as an *int*.

The JVM presents an interesting challenge as far as the effective address of load and store instructions is concerned. Because the JVM is a stack-based machine, it is not clear what the “effective” address of a load or store instruction would be since the JVM does load and store for local variables by indexing in its stack. More importantly, the JVM hides both physical and virtual addresses from bytecodes. As an example, consider a simple Java program shown in Figure 5, which gets compiled into bytecodes shown in Figure 6 by Sun Microsystem Inc.’s `javac` compiler.

```
public class address {
    public int i;

    // constructor
    public address(int x) {
        i = x;
    }

    public static void main(String a[]) {
        address a;

        a = new address(5);
        System.out.println(a.i);
    }
}
```

Figure 5. A Sample Java Program that Accesses a Member Variable

As shown in these figures, the JVM never addresses any objects directly, but it performs operations on objects via values of type **reference**. When a new object is created, for example, a reference to the newly created object is pushed onto the stack. Unlike pointers to objects, which point to the object directly, a reference to an object in Sun's current implementation of the JVM is a pointer to a handle (Lindholm and Yellin 1997). This handle consists of two pointers that point to the memory allocated for the object data and to a table containing the object's methods.

```
public class address {
    public int i;

    // constructor
    public address(int x) {
        aload_0
        // call superclass's constructor
        invokenonvirtual    Object.init
        aload_0
        iload_1
        putfield            address.i
        return
    }

    public static void main(String a[]) {
        // create new address class
        new                address
        // a reference has been created on the stack
        dup
        iconst_5
        // call constructor
        invokenonvirtual    address.init
        pstore_1
        // the rest of the code is for invoking println()
        getstatic           java/lang/System.out
        aload_1
        getfield            address.i
        invokevirtual       java/lang/System.println
        return
    }
}
```

Figure 6. A Compiled Java Program that Accesses a Member Variable

Therefore, methods for dealing with the effective address of an instruction are not provided in the current implementation of BIT. However, if there is an implementation of the JVM that does not hide the effective address of an instruction, these methods could be added to the `Instruction` class. Another approach to overcoming this limitation would be to implement the JVM in analysis methods to obtain the addresses of loads and stores.

CHAPTER V

BIT IMPLEMENTATION

This chapter presents some of the issues that arose when we implemented BIT including an implementation rationale and the details on how we added calls to analysis methods in bytecodes.

RATIONALE

There were many different approaches that could have been taken to observe and measure the dynamic behavior of Java bytecodes. One possible approach was to modify the JVM to produce relevant outputs. However, this meant that each time we wanted to create a customized tool, we would have to dig inside the JVM to add or remove tracing code. This would have resulted in unnecessary complexity. An even bigger problem would be that even if we had modified the JVM significantly to produce very meaningful traces, we would not be able to redistribute it for others to use because of license agreements that would have restricted the redistribution of the JVM. Furthermore, tracking changes made to the JVM implementation would be difficult.

This project started following EEL's design because of its claims of portability, but we soon realized it was too complex for what we wanted to do. EEL was designed for the SPARC architecture, which is much more complex than the JVM, and because EEL needed to handle such architectural features as delayed branches and indirect calls, it proved to be overkill for the JVM. Using EEL also meant porting or writing something equivalent to the GNU *bfd* library (Chamberlain

1991), which EEL used. EEL's object oriented design was still followed. However, we modeled BIT after ATOM for analysis and instrumentation, which proved to be simpler for both the users and the designers of the system.

BIT is compatible with Just-In-Time (JIT) translators since it modifies bytecodes before they are translated by JIT translators. JIT translators try to speed the execution of JVM programs by translating bytecode instructions into native machine instructions on the fly and reuse the native machine instructions if the same bytecodes are executed more than once.

INTERMEDIATE REPRESENTATION

BIT consists of two distinct Java packages: one for performing low-level operations such as reading and writing class files, interpreting constant pool entries, reading code buffers, and other low-level class file parsing. A second package is used for performing higher-level operations such as finding and constructing basic blocks, decoding instructions from the code buffer, inserting method calls, and navigating through higher intermediate representations. The first package provides a low-level representation of a class file while the second package builds on top of the first one to provide higher-level functionality. This approach was taken purposely to make incremental development possible since we did not know how this project would turn out at the beginning. (Lindholm and Yellin 1997) provides very specific details on how to interpret a `class` file and presents a representation of a `class` file. Rather than reinventing the wheel, their representation of a class file was used.

ADDING METHOD CALLS

To add a method call before or after a certain entity, the descriptor, the class name, and the method name of the method being inserted are added to the constant pool table unless they are already present because they were either added earlier by the same method or by another method being inserted. This way, the constant pool table, which is limited to 65535 entries, is saved in case there are more entries to be inserted. The constant pool table is also used as a place to store arguments to the analysis methods unless the argument is a String object whose value is “BranchOutcome.” In this case, appropriate bytecode instructions are added to obtain the outcome of the branch at run-time and passed to the analysis method.

Analysis method calls are inserted by using the `invokestatic` bytecode instruction and therefore, analysis methods have to be `static`. This also implies that objects cannot be associated with these methods. Other method invocation bytecodes such as `invokevirtual` could have been used as well, but to keep things simple, only `invokestatic` instruction is used. To use `invokevirtual`, more complex sequences of bytecodes would have to be inserted in the instrumented program because an instance of the class would need to be created and manipulated.

Being able to pass more than one argument to an analysis method is a big plus, and that ability will soon be added to BIT. To pass more than one argument to an analysis method, one would have to know the descriptor of that analysis method in advance; this could be accomplished by sending the descriptor to the instrumentation code beforehand by using a method call similar to the `AddProc()` procedure in ATOM (Srivastava and Eustace 1994). A descriptor in Java is a string that represents

the type of a method. For example, the method descriptor for the method `void foo(Object x, int y)` is **(Ljava/lang/Object;I)V**. The characters enclosed in the parenthesis are the parameter descriptors. An **L** preceding **java/lang/Object** indicates that the first parameter has the type of an instance of the class **java/lang/Object**, and **I** indicates that the second parameter is an integer type. The characters after the closing parenthesis describe the return descriptors. In this case, a **V** is used to indicate that the method returns no value.

CHAPTER VI

EXAMPLE APPLICATIONS

This chapter presents two example tools implemented using BIT and their performance on three Java applications: a benchmark suite, a lexical analyzer generator, and BIT itself.

Several small tools were written to exercise BIT, which include a branch counting tool and a dynamic instruction counting tool. The branch counting tool was built by inserting calls to analysis methods before all bytecode conditional instructions in the user program (see Figure 2). These analysis methods were passed the value of the program counter and the outcome of the branch. The analysis methods then printed the statistics associated with a particular branch. The dynamic instruction counting tool was built by inserting calls to analysis methods before basic blocks. The analysis method was passed the sizes of the basic blocks, which were added and printed to show how many JVM instructions were executed during the course of the user program's execution. The actual code for the dynamic instruction counting tool is included in appendix A.

PERFORMANCE

To learn about the performance of BIT, three characteristics were measured on an Intel Pentium 166Mhz machine with 24 MB of memory running Microsoft Windows 95 and Sun Microsystems Inc.'s Java Development Kit (JDK) version 1.1.2. The characteristics measured were time required to instrument user programs, execution time of the instrumented programs, and the size of instrumented programs.

For these measurements, Jmark 1.01 (Dragan and Seltzer 1996), a JVM benchmark suite from Ziff-Davis publishing company, JLex (Berk 1997), a lexical analyzer generator for Java, and BIT were used as user applications on which the custom tools mentioned above were run. Jmark consists of 19 class files and benchmarks 11 different areas, JLex consists of 20 class files, and BIT consists of 43 class files. Table 1 summarizes the time taken for each tool to build the instrumented programs.

Table 1. Time Required to Instrument User Programs

APPLICATION	BRANCH COUNT	DYNAMIC INSTRUCTION
Jmark	68 seconds	84 seconds
JLex	74 seconds	96 seconds
BIT	149 seconds	176 seconds

As shown in Table 1, the time taken to instrument each of the three applications was under three minutes for both of the customized tools. The average time taken to instrument a single class file was about four seconds. BIT has more classes and has larger code size, and this explains why it took more time to instrument BIT than the other two applications.

Table 2 shows the execution time of the instrumented programs for each tool.

Table 2. Execution Time of Instrument User Programs

APPLICATION	UNINSTRUMENTED	BRANCH COUNT (raw / % increase over uninstrumented)	DYNAMIC INSTRUCTION (raw / % increase over uninstrumented)
Jmark	345 seconds	415 seconds / 20%	396 seconds / 15%
JLex	73 seconds	243 seconds / 233%	215 seconds / 195%
BIT	68 seconds	198 seconds / 191%	178 seconds / 162%

The execution time of the instrumented program was increased, and the increase ranged from 15% to 233%. This increase in the execution time is due to method invocation overhead when invoking analysis methods and the time actually spent in the analysis methods. There was a wide variation on the percentage increase of the execution time, and we believe that this is due to the fact that there are more bytecodes that get instrumented in JLex and BIT (86,341 bytes and 96,388 bytes respectively as opposed to 34,966 bytes in Jmark), and the analysis methods get invoked more frequently. These frequent analysis method invocations, in turn, lead to increased file accesses that slow the execution down considerably. We measured the dynamic frequency of conditional branch instructions in JLex and BIT and found that 11.4% of all instructions were conditional branches in JLex while 6.8% of all instructions were conditional branches in BIT. Because conditional instructions are more frequent in JLex, there are more analysis method invocations and a higher instrumentation overhead in JLex.

The increase in the execution time of the program instrumented with the branch counting tool was higher than that of the program instrumented with the dynamic instruction counting tool because the former has to do more record keeping in the analysis methods. Since there are more analysis method invocations in the program instrumented with the dynamic instruction counting tool (there is one method invocation for each basic block that get executed), the record keeping in the program instrumented with branch counting tool obviously outweighs the method invocation overhead (there is one method invocation for each conditional basic block that gets executed). To verify this, we created other versions of branch and dynamic

instruction counting tools that had same instrumentation code but whose analysis code was empty. The execution time of the program instrumented with the modified branch counting tool was indeed shorter than that of the program instrumented with the modified dynamic instruction counting tool.

Table 3. Code Size of Instrument User Programs

APPLICATION	UNINSTRUMENTED	BRANCH COUNT (raw / % increase over uninstrumented)	DYNAMIC INSTRUCTION (raw / % increase over uninstrumented)
Jmark	34,966 bytes	44,130 bytes / 26%	47,854 bytes / 37%
JLex	86,341 bytes	106,996 bytes / 24%	110,344 bytes / 28%
BIT	96,388 bytes	109,701 bytes / 14%	116,054 bytes / 20%

There was an increase in the program size as well, which ranged from 14% to 37% as shown in Table 3. This increase in size is explained by the addition of entries in the constant pool table, which include static arguments to the analysis routine, the names of class and analysis methods, and method descriptors, and by the addition of actual bytecodes to invoke the analysis routines. The size of the program instrumented with the dynamic instruction counting tool increased more than that of the program instrumented with the branch counting tool because the former includes bytecodes to invoke the analysis routines before each basic block while the latter only includes bytecodes to invoke the analysis routine before conditional instructions.

IMPROVING PERFORMANCE

Now that we have described the performance measurements, suggestions on how to improve BIT's performance is in order. To reduce the time required to instrument user programs, we could use arrays instead of vectors since vectors in Java

tend to be two to three times slower than arrays according to our personal experiences. Also, when BIT is compiled into native machine code by using one of the native compilers for Java, its instrumenting speed should increase considerably. To increase the execution speed of the instrumented user programs, we could inline analysis methods. Doing so would increase the execution speed by removing method invocation overhead as in (Srivastava and Eustace 1994). The execution speed of the instrumented user programs should also increase when they can be translated into native machine code and executed directly.

CHAPTER VIII

SUMMARY

BIT is a set of interfaces that brings the functionality of ATOM and other related tracing tools to the Java world by allowing a user to instrument a class file by inserting calls to analysis methods. There are areas where being able to create customized tools to observe and measure the run-time behavior of programs is very valuable such as for optimizations, simulations, and measuring the effectiveness of different algorithms. To design and implement BIT, various approaches to instrumenting binaries were examined and investigated. Also, we examined the inner workings of the JVM and its bytecode format. BIT allows the user to enter calls to analysis methods in the `class` file to obtain dynamic information about the program.

BIT is the first framework that allows the users to create customized tools to analyze JVM bytecodes fast and easily. A performance study was also conducted for small applications that use BIT, and we reported there results here. The overheads for both the code size and the execution time for a smaller program were between 15% to 37%. However, the overhead for the execution time for a larger program was about two times the original execution time.

We believe that BIT is an effective framework for understanding the dynamic behavior of JVM bytecodes, and we hope it will serve its users well in their studies related to the JVM and JVM bytecodes.

There are issues that need to be addressed in the near future, and in the rest of this chapter, we discuss how they might be addressed.

One issue is the handling of exceptions. An exception in Java bytecode contains information about the exception handler in the code buffer, but since BIT changes the code buffer as a result of adding method calls, the information about the exception handler in an exception is no longer valid. This could result in run-time errors since incorrect exception handlers could be invoked when exceptions are raised. Checks are needed to make sure that the information about the exception handlers are also updated if they are going to be affected by changes in the code buffer. Exceptions are being ignored in the current implementation.

Another issue involves the use of the constant pool table as a place for argument passing. This table could overflow if too many methods are being inserted. To avoid possible constant table overflow, it might be better to use local variable space. However, this will require more complexity in adding bytecode instructions to the code buffer since depending on the values of the arguments, different bytecode instructions would need to be generated and inserted to the code buffers.

As it was previously described, being able to pass more than one argument to the analysis methods is a big plus, and this could be accomplished by sending the type description of the method beforehand by using a method call similar to the **AddProc()** procedure in ATOM (Srivastava and Eustace 1994).

Larger customized tools using BIT need to be built to prove BIT's usefulness. A possible candidate includes a tool that performs hierarchical profiling of a class file such as gprof (Graham, Kessler, and McKusick 1983), HiProf (Garvin), and mprof (Zorn and Hilfinger 1988). Recently, there have been other advances in profiling including flow and context sensitive profiling (Ammons, Ball, and Larus 1997) and

interprocedural dataflow analysis (Goodwin 1997). These advanced profiling techniques could also be applied to JVM programs with BIT.

Furthermore, the main focus when implementing BIT was on functionality and not on performance. Therefore, optimizing BIT to improve its performance is also in order.

REFERENCES

- Anant Agarwal, Richard L. Sites and Mark Horowitz. ATUM: A New Technique for Capturing Address Traces Using Microcode. *Proceedings of the 13th International Symposium on Computer Architecture*, pages 199-127, June 1986.
- Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 85-108, June 1997.
- Elliot Berk. JLex: A Lexical Analyzer Generator for Java.
<http://www.cs.princeton.edu/~appel/modern/java/JLex>.
- Brian Bershad et al. Etch Overview.
<http://www.cs.washington.edu/~bershad/Etch.html>.
- Eric A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook and William E. Weihl. PROTEUS: A High-Performance Parallel-Architecture Simulator. Massachusetts Institute of Technology technical report MIT/LCS/TR-516, 1991.
- Steve Chamberlain. *Libbfd: The Binary File Descriptor Library*. Cygnus Support, bfd version 3.0 edition, April 1991.
- Douglas Clark. Cache Performance in the VAX 11/780. *ACM Trans. Computer Systems*, vol 1, no 1, February 1983.
- Robert F. Cmelik and David Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 128-137, May 1994.
- Richard V. Dragan and Larry Seltzer. Java Speed Trials. *PC Magazine*, vol 15, no 18, 1996.
- Susan J. Eggers, David R. Keppel, Eric J. Koldinger, and Henry M. Levy. Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor. *SIGMETRICS Conference on Measurement and modeling of Computer Systems*, vol 8, no 1, May 1990.
- Janel Garvin. HiProf Advanced Code Performance Analysis Through Hierarchical Profiling.
<http://tracepoint.galatia.com/noframes/products/hiprof/profiling/overview>.
- David W. Goodwin. Interprocedural Dataflow Analysis in an Executable Optimizer. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 122-145, June 1997.

- James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.
- Susan L. Graham, Peter B. Kessler and Marshall K. McKusick. An Execution Profiler for Modular Programs. *Software Practice and Experience*, pages 671-685, vol 13, 1983.
- Amir H. Hashemi, David R. Kaeli, and Brad Calder. Efficient Procedure Mapping Using Cache Line Coloring. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 171-182, June 1997.
- Reed Hastings and Bob Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. *Proceedings of the Winter USENIX Conference*, Pages 125-136, January 1992.
- James R. Larus and Thomas Ball. Rewriting Executable Files to Measure Program Behavior. *Software, Practice and Experience*, vol 24, no. 2, pages 197-218, February 1994.
- James R. Larus and Eric Schnarr. EEL: Machine-Independent Executable Editing. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, pages 291-300, June 1995.
- Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.
- MIPS Computer Systems, Inc. *Assembly Language Programmer's Guide*, 1986.
- J. Eliot B. Moss and Antony L. Hosking. Approaches to Adding Persistence to Java. *First International Workshop on Persistence and Java*, September 1996.
- ODI. *PSE/PSE Pro for Java API User Guide*. 1997.
- Todd A. Proebsting. Simple Translation of Goal-Directed Evaluation. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1-6, June 1997.
- Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 48-60, May 1993.
- Michael D. Smith. Tracing with Pixie. Memo from Center for Integrated Systems, Stanford Univ., April 1991.

- K. So et al. PSIMUL – A System for Parallel Execution of Parallel Programs. in *Performance Evaluation of Supercomputers*, J.L. Martin, ed., Elsevier Science Publishers B.V., North Hoolan, 1988.
- Amitabh Srivastava and Alan Eustace. ATOM A System for Building Customized Program Analysis Tools. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI)*, pages 196-205, June 1994.
- Amitabh Srivastava and David Wall. Link-Time Optimization of Address Calculation on a 64-bit Architecture. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI)*, pages 49-60, June 1994.
- Amitabh Srivastava and David Wall. A Practical System for Intermodule Code Optimization at Link-Time. *Journal of Programming Languages*, vol 1, no 1, pages 1-18, March 1993.
- David W. Wall. Systems for Late Code modification. In Robert Giegerich and Susan L. Graham, eds., *Code Generation – Concepts, Tools, Techniques*, pages 275-293, Springer-Verlag, 1992.
- Benjamin Zorn and Paul Hilfinger. A Memory Allocation Profiler for C and Lisp Programs. *USENIX Conference Proceedings*, pages 223-237, Summer 1988.

APPENDIX

APPENDIX A. ICount.java

```
// ICount.java
// This program outputs the number of instructions that got executed.
//
// A sample program that exercises the ClassFileVisit package.
//
// Copyright (c) 1997 by Han B. Lee (hanlee@cs.colorado.edu).
// ALL RIGHTS RESERVED.
//
// Permission to use, copy, modify, and distribute this software and its
// documentation for non-commercial purposes is hereby granted provided
// that this copyright notice appears in all copies.
//
// This software is provided "as is". The licensor makes no warranties, either
// expressed or implied, about its correctness or performance. The licensor
// shall not be liable for any damages suffered as a result of using
// and modifying this software.

import classFileVisit.*;
import java.io.*;
import java.util.*;

public class ICount {
    private static PrintStream out = null;
    private static int i_count = 0;

    public static void main(String argv[]) {
        try {
            // if wrong syntax
            if (argv.length != 2) {
                System.out.println("Syntax: java ICount <inclassfilename>
                                   <outclassfilename>");
                System.out.println("example: java ICount foo.class bar.class");
                return;

                // otherwise, get the filename
                String infilename = new String(argv[0]);
                String outfilename = new String(argv[1]);

                // create class info object
                ClassInfo ci = new ClassInfo(infilename);

                // get the routines defines in this class
                Vector routines = ci.getRoutines();

                // loop through all the routines
                // see java.util.Vector for more information on Vector class
                for (Enumeration e = routines.elements(); e.hasMoreElements(); ) {
                    // get a routine
```

```

        Routine routine = (Routine) e.nextElement();
        // analyze routine's code
        // this method needs to be invoked before accessing instructions
        routine.analyzeCode();
        routine.findBasicBlocks();

        Vector instructions = routine.getInstructions();

        for (Enumeration b = routine.getBasicBlocks().elements();
             b.hasMoreElements(); ) {
            BasicBlock bb = (BasicBlock) b.nextElement();
            bb.addBefore("ICount", "count", new Integer(bb.size()));
        }

        // get the name of the method
        String method_name = routine.getMethodName();
        String class_name = ci.getClassName();

        routine.addBefore("ICount", "openStream", class_name);
        routine.addAfter("ICount", "printICount", method_name);
    } // end of routine iteration

    ci.write(outfilename);
}

catch (Exception e) {
    System.out.println(e.getMessage());
}
}

public static void openStream(String s) {
    try {
        if (out == null)
            out = new PrintStream(new FileOutputStream(s + ".icount"));
    }
    catch (Exception e) {
        System.out.println(e.getMessage());
    }
}

public static void printICount(String s) {
    try {
        System.out.println(i_count + " instructions executed in method " + s + ".");
        i_count = 0;
    }
    catch (Exception e) {
        System.out.println(e.getMessage());
    }
}

public static void count(int incr) {
    i_count+=incr;
}
}

```