BIT: A Tool for Instrumenting Java Bytecodes

Han Bok Lee and Benjamin G. Zorn
University of Colorado, Boulder

BIT: A Tool for Instrumenting Java Bytecodes[1]

Han Bok Lee
hanlee@cs.colorado.edu
Department of Computer Science
University of Colorado, Boulder 80309
Benjamin G. Zorn
zorn@cs.colorado.edu
Department of Computer Science
University of Colorado, Boulder 80309

Abstract

BIT (Bytecode Instrumenting Tool) is a collection of Java classes that allow one to build customized tools to instrument Java Virtual Machine (JVM) bytecodes. Because understanding program behavior is an essential part of developing effective optimization algorithms, researchers and software developers have built numerous tools that carry out program analysis. Although there are existing 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 JVM bytecodes. In this paper, we describe BIT, which 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. In this paper, we describe several simple tools built using BIT and also report on BIT's performance. We found that the overhead for the execution speed and size were between 23% to 150%.

1. Introduction

It is often important for software developers and researchers to be able to measure and understand both the static structure and dynamic behavior of a program. Such information is used to identify critical pieces of code; for debugging purposes; to evaluate and compare the performance of different software or hardware implementations such as branch prediction, cache replacement, and instruction scheduling; and in support of profile-driven optimizations [31]. Over the years, researchers have built numerous tools that allow them to obtain this information.

This paper describes BIT (Bytecode Instrumenting Tool) [20], a tool that allows JVM bytecodes to be instrumented for the purpose of extracting measurements of their dynamic behavior. The JVM is an abstract machine specification designed to support the Java programming language, and JVM bytecodes are equivalent to binaries on other machines [21]. Although there are tools that allow binary instrumentation on a number of different operating systems and machine architectures, BIT is the first framework of which we are aware that supports JVM bytecode instrumentation. BIT is a set of Java classes that allow the user to observe the dynamic behavior of programs by inserting calls to user analysis methods at any point in the bytecode execution. Because BIT is written in Java, tools written using it are portable across platforms. Also, there are other programming languages that can be compiled into JVM bytecodes such as Kawa [6], which compiles Scheme code into JVM bytecodes and AppletMagic [17], which translates Ada 95 to JVM bytecodes. Furthermore, Hardwick and Sipelstein studied the feasibility of using Java as an intermediate language [14]. Because BIT instruments JVM bytecodes, it can be used to instrument programs written in any language that has been compiled to the JVM, and instrumentation does not require that the program source code be available.

In this paper we describe the design and implementation of BIT, present example tools built using BIT, and describe results based on measuring BIT's overhead on five Java programs, including BIT itself. BIT's instrumentation makes the instrumented JVM executables both larger and longer running. We found that the overhead for the execution speed and code size ranged from 23% to 150%.

This paper has the following organization. In Section 2, we discuss related work. In Section 3, we describe BIT's design and introduce a sample tool written using

[1] This work was supported in part by the National Science Foundation under grant IRI-95-21046.

BIT. In Section 4, we discuss some details of the implementation and in Section 5 we present performance results based on instrumenting several Java applications. Finally, in Section 6, we summarize the paper and discuss future directions for research.

2. Related Work

There are many tools that employ techniques based on program instrumentation to carry out different tasks. These tasks range from emulation and tracing to optimization [19]. The Wisconsin Wind Tunnel architecture simulator [27], for example, allows the emulation of a cycle counter, which the underlying hardware does not provide. Techniques based on program instrumentation have also been used in optimizations [31, 32]. However, most of the tools that have been developed using program instrumentation techniques are used for studying program or system behavior. Tools such as QPT [18], Pixie [28], and Epoxie [33] generate address traces and instruction counts by rewriting program executables. MPTRACE [13] and ATUM [1] generate data and instruction traces, and PROTEUS [5] and Shade [7] emulate other architectures. Also, software testing and quality assurance tools that detect memory leaks and access errors such as Purify [15] 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 of 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 [30].

There is another group of tools, sometimes called binary editing or executable editing tools, which have different design goals and offer a library of routines for modifying executable files. The tools in this group include the OM system [32], EEL [19], ATOM [30], and Etch [4], which are explained in more detail below. 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 Register Transfer language (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 of 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 program 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 constructs and inserting procedure calls before and after them. However, 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 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 the *prof* switch. There is a tool called NetProf [26] that visualizes Java profile information by translating Java bytecodes to Java source code. There are also tools that carry out post-processing on class files such as osjcfp [25] and the work by Cattell [23] 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 follows ATOM's design by providing a set of classes that users can employ to build their own program analysis tools for JVM bytecodes.

3. BIT Architecture

This section describes the design of BIT at a high level and illustrates how BIT is used with an example. 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.

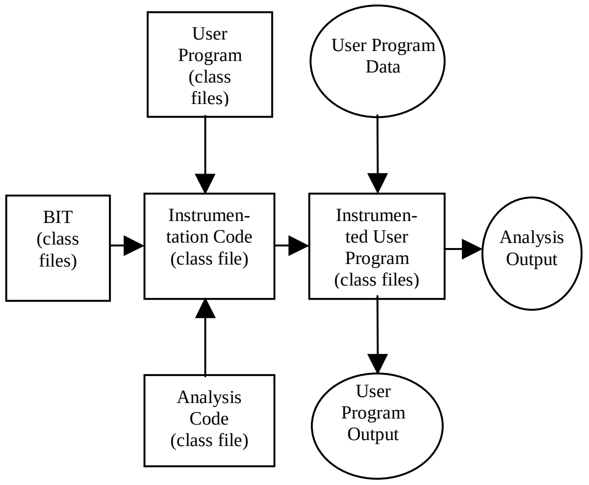
User Program (class files)
User Program Data
BIT (class files)
Instrumen-tation Code (class file)
Instrumen-ted User Program (class files)
Analysis Output
Analysis Code (class file)
User Program Output

Figure 1. The Process of Using BIT – Instrumentation Code uses BIT classes to read in User Program and to insert calls to Analysis Code to produce Instrumented User Program.

Figure 1 illustrates the process of using BIT. BIT is a set of Java classes that are represented by the BIT box. The user's application is compiled into JVM bytecodes with a Java compiler. The User Program box represents the application output. The user writes personalized instrumentation code by using the classes and methods that BIT provides. The user also writes analysis code. Both instrumentation and analysis code are compiled into JVM bytecodes by a Java compiler and are represented by Instrumentation and Analysis Code boxes respectively. When the JVM executes the instrumentation code, it will read in the user program, which is in the form of JVM bytecodes, and insert calls to the 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 both the original program output, which is represented by the User Program Output circle, and the analysis output, which is represented by the Analysis Output circle. The inserted calls to the analysis routines do not have any semantic effect on the instrumented program, which should produce exactly the same output as the original program.

To provide a concrete understanding of how BIT works, we present a customized tool that could aid in branch prediction as illustrated by Srivastava and Eustace [30]. This tool counts the number of branches taken and not taken at all the branches in different methods. Figure 2 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.

Once we find conditional instructions, we 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.

Figure 3 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. The analysis code uses class variables *branch* and *pc*

```
import BIT.*;

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 (Enumeration e=routines.elements();e.hasMoreElements(); ){
            Routine routine = (Routine) e.nextElement();
            Vector instructions = routine.getInstructions();
            for (Enumeration b = routine.getBasicBlocks().elements(); b.hasMoreElements(); ) {
                BasicBlock bb = (BasicBlock) b.nextElement();
                Instruction instr = (Instruction)instructions.elementAt(bb.getEndAddress());
                short instr_type = InstructionTable.InstructionTypeTable[instr.getOpcode()];
               if (instr_type == InstructionTable.CONDITIONAL_INSTRUCTION) {
                 instr.addBefore("BranchPrediction", "Offset", new Integer(instr.getOrigOffset()));
                 instr.addBefore("BranchPrediction", "Branch", new String("BranchOutcome"));
               }
            }
            String method = new String(routine.getMethod());
            routine.addBefore("BranchPrediction", "EnterMethod", method);
            routine.addAfter("BranchPrediction", "LeaveMethod", method);
        }
        ci.write(outfilename);
    }
}
```

Figure 2. Instrumentation Code: Branch Counting Tool

```
public BranchPrediction {
    static Hashtable branch = null;
    static int pc = 0;

    public static void EnterMethod(String s) {
        System.out.println("method: " + s);
        branch = new Hashtable();
    }

    public static void LeaveMethod(String s) {
        System.out.println("stat for method: " + s);
        for (Enumeration e = branch.keys(); e.hasMoreElements(); ) {
            Integer key = (Integer) e.nextElement();
            Branch b = (Branch) branch.get(key);
            int total = b.taken + b.not_taken;
            System.out.print("PC: " + key);
            System.out.print("\t\ttaken: " + b.taken + " (" + b.taken*100/total + "%)");
            System.out.println("\t\tnot taken: " + b.not_taken + " (" + b.not_taken*100/total + "%)");
        }
    }

    public static void Offset(int offset) {
        pc = offset;
    }

    public static void Branch(int brOutcome) {
        Integer n = new Integer(pc);
        Branch b = (Branch) branch.get(n);
        if (b == null)
          b = new Branch();
        if (brOutcome == 0)
           b.taken++;
        else
          b.not_taken++;
}
```

Figure 3. Analysis Code: Branch Counting Tool

because the current version of BIT does not allow passing more than one argument to the analysis methods at this time, a shortcoming that will be fixed in future work.

Analysis code is likely to vary between different customized tools depending on what their functional requirements are. However, most of the instrumentation code presented in Figure 2 is likely to be the same for different customized tools since all of them will require some sort of navigation through different constructs within a class file. For instance, to dynamically count the number of instructions that get executed in a user program, we would only have to change the body of the loop that obtains different basic blocks as shown in Figure 4. Instead of checking whether the last instruction in a basic block is a conditional or not, we count the number of instructions present in a basic block to obtain the total instruction count.

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

Figure 4. Changes Required to Instrumentation Code to Count Instructions instead of Conditionals

4. BIT Implementation

There were several different approaches that could have been taken to observe and measure the dynamic behavior of Java bytecodes. One possible approach would be to modify the JVM to produce relevant outputs. An advantage of this approach is that it is easier to obtain certain kinds of information that would either be difficult or impossible to obtain otherwise. For example, measurements of the JVM garbage collection implementation would require JVM modifications. A drawback of this approach is that each time we wanted to create a customized tool, we would have had revisit the JVM source to add or remove tracing code. An even bigger problem would be that even if we modified the JVM source, redistributing the tools would be difficult due to licensing restrictions. Furthermore, tracking changes made to the JVM implementation, as new releases became available, would also be difficult.

We originally considered using EEL as a basis for our design, but we concluded that EEL supported more functionality than we required. EEL was designed for the SPARC architecture, which is more complex than the JVM, and it proved to be overkill for instrumenting JVM bytecodes. Furthermore, developing the BIT system in Java allowed us to create highly portable instrumentation tools. Although EEL's object-oriented design was still followed, we modeled BIT's functionality after that of ATOM (i.e., only allowing executable instrumenting and not arbitrary editing), which we felt was still highly valuable and much easier to design, implement, and use.

4.1. Adding Method Calls

Since the instrumentation process requires adding new method calls to a class file, class and method names as well as other constants about these methods need to be inserted to the *constant pool table*. The *constant pool table* is a place where different string constants, class names, field names, and other constants are stored for each class file. To support the ability to add a method call before or after a certain entity (e.g., method, basic block, etc.), the descriptor, the class name, and the method name of the method being inserted need to be present in the constant pool table of the code being instrumented. The constant pool table is also used as a place to store arguments to the analysis methods.

If the string "BranchOutcome" is used as an argument to the analysis methods, then it is interpreted as a special directive for obtaining the outcome of the branch instruction since there is no way of knowing what the outcome of the branch would be at instrumentation time. In this case, appropriate bytecode instructions are added to obtain the outcome of the branch at run-time and pass it to the analysis method.

In the JVM, there are several method invocation instructions such as *invokestatic*, *invokevirtual*, *invokespecial*, and *invokeinterface*. In BIT, analysis method calls are inserted by using the *invokestatic* bytecode instruction and therefore, analysis methods have to be static. This decision implies that objects cannot be associated with these methods. The *invokevirtual* bytecode could have been used, but to keep things simple, only the *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.

5. Performance Results

In this section, we present results based on applying two example tools implemented using BIT to five Java applications: a benchmark suite, a lexical analyzer generator, a Java compiler, an LALR parser generator, and BIT itself.

We wrote several small tools, such as a branch counting tool and a dynamic instruction counting tool, to exercise BIT. The branch counting tool is a version of the tool presented in Figures 2 and 3, modified to produce less output. We built the dynamic instruction counting tool by inserting calls to analysis methods before basic blocks. The analysis method receives the sizes of the basic blocks and adds and prints them to show how many JVM instructions were executed during the course of the user program's execution. For the results presented here, we instrumented only the application code (i.e., not the Java library classes). Had we instrumented the library classes as well, the overheads would undoubtedly be higher.

To learn about the performance of BIT, three characteristics were measured on an Intel Pentium 200Mhz machine with 24 MB of memory running Microsoft Windows 95 and Sun Microsystems Inc.'s Java Development Kit (JDK) version 1.1.4. 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 1.2.1 [8], a JVM benchmark suite from Ziff-Davis publishing company; JLex [3], a lexical analyzer generator for Java; EspressoGrinder [24], a Java compiler; CUP [16], a parser generator; 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 of Java performance, JLex consists of 23 class files, EspressoGrinder is composed of 105 class files, CUP consists of 40 class files, and BIT consists of 43 class files.

Table 1 summarizes the time taken for each tool to build the instrumented programs. As shown in Table 1, the time taken to instrument each of the five applications was under four minutes for both of the customized tools. The average time taken to instrument a single class file was about two seconds. EspressoGrinder has more classes and a larger code size, and this explains why it took more time to instrument EspressoGrinder than the other four applications. As native code compilers become available and we are able to compile BIT, the time required to instrument applications should decrease significantly. The first column in this table is the time required to compile Java files using *javac* compiler and is included to aid the reader in estimating cost of instrumenting class files relative to compilation. In the cases where we did not have the program sources, the compilation time is indicated as N/A.

Table 1. Time Required to Instrument User Programs

APPLICATION	COMPI-LATION TIME (seconds)	BRANCH COUNT INST. TIME (seconds)	DYNAMIC INSTRUCTION INST. TIME (seconds)
Jmark	N/A	14	17
JLex	15	89	89
EspressoGrinder	N/A	233	174
CUP	24	55	88
BIT	15	32	31

Table 2 shows the execution time of the instrumented programs for each tool. The increase in execution time of the instrumented programs ranged from 23% to 150%. This increase is due to method invocation overhead when invoking analysis methods and the time actually spent in the analysis methods. For most of the programs, the execution time was increased by approximately a factor of one to one and a half. The lower overhead observed in Jmark is most likely due to its extensive use of the Java libraries (e.g., for the purpose of benchmarking graphics, etc.), which were not instrumented in this study.

Table 2. Execution Time of Instrumented User Programs

APPLI-CATION	UN-INSTRU-MENTED	BRANCH COUNT (raw/% increase over uninstrumented)	DYNAMIC INSTRUCTION (raw / % increase over uninstrumented)
Jmark	315 seconds	409 seconds / 30%	387 seconds / 23%
JLex	16 seconds	31 seconds / 94%	20 seconds / 25%
Espresso Grinder	6 seconds	15 seconds / 150%	12 seconds / 100%
CUP	7 seconds	14 seconds / 100%	8 seconds / 14%
BIT	89 seconds	183 seconds / 106%	167 seconds / 88%

To get a better understanding of the programs we instrumented, we also measured some of the basic dynamic characteristics of the programs. In particular, we looked at the average bytecode instructions executed per basic block. This average (computed dynamically) was measured to be 4.4 in JLex, 5.8 in EspressoGrinder, 3.2 in CUP, and 3.7 in BIT.

We also measured the dynamic frequency of conditional branch instructions in JLex, EspressoGrinder, CUP, and BIT, and observed that 11.4% of all instructions were conditional branches in JLex; 10.3% in EspressoGrinder; 12.9% in CUP; and 6.8% in BIT.

BIT instrumentation caused an increase in the program size as well, and the overhead 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 includes static arguments to the analysis routines, 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 except for one application. This is 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. However, since EspressoGrinder has a relatively large number of instructions per basic block compared to the other applications, the branch counting tool had more overhead in this application.

Table 3. Code Size of Instrumented User Programs

APPLI-CATION	UN-INSTRU MENTED	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	86,350 bytes	107,869 bytes / 25%	111,173 bytes / 29%
Espresso Grinder	295,281 bytes	379,879 bytes / 29%	374,460 bytes / 27%
CUP	117,964 bytes	147,738 bytes / 25%	151,933 / 29%
BIT	95,680 bytes	108,404 bytes / 13%	110,953 bytes / 16%

The results presented show that the prototype version of BIT has acceptable performance for several instrumentation tasks. We anticipate that the overheads of the prototype can be reduced dramatically by making a number of performance enhancements. 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. To increase the execution speed of the instrumented user programs, we could inline analysis methods, removing method invocation overhead (see [30]). Allowing multiple arguments to analysis routines would significantly decrease the number of method calls required to do the instrumentation, and would also substantially improve performance.

6. 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 JVM class file. Being able to create customized tools to observe and measure the run-time behavior of programs is valuable for many tasks including program optimization and system design.

BIT is the first framework that allows users to create customized tools to analyze JVM bytecodes quickly and easily. BIT allows the user to add calls to analysis methods any place in the JVM bytecode to obtain dynamic information about the program. We conducted a performance study based on two small tools written using BIT, and we reported the results here. The overheads for both the code size and the execution time were between 23% to 150%.

There are issues that need to be addressed in the near future, and in the rest of this section, we discuss them. 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.

Larger customized tools using BIT need to be built to prove BIT's usefulness. Possible candidates include a tool that performs hierarchical profiling of a class file such as gprof [13], HiProf [10], and mprof [34]. Recently, there have been other advances in profiling including flow and context sensitive profiling [2] and interprocedural dataflow analysis [11]. These advanced profiling techniques could also be applied to JVM programs with BIT.

7. Availability

BIT source is freely available. If you would like to obtain a copy, please email hanlee@cs.colorado.edu or zorn@cs.colorado.edu.

Acknowledgments

This work was supported in part by the National Science Foundation under grant IRI-95-21046. We would like to thank Margo Seltzer and the anonymous referees for their helpful comments on drafts of this paper.

References

[1] 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.

[2] 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.

[3] Elliot Berk. JLex: A Lexical Analyzer Generator for Java. http://www.cs.princeton.edu/~appel/modern/java/JLex.

[4] Brian Bershad et al. Etch Overview. http://www.cs.washington.edu/~bershad/Etch.html.

[5] 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.

[6] Per Bothner. Kawa, the Java-based Scheme System. http://www.cygnus.com/~bothner/kawa.html.

[7] 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.

[8] Richard V. Dragan and Larry Seltzer. Java Speed Trials. *PC Magazine*, vol 15, no 18, 1996.

[9] 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.

[10] Janel Garvin. HiProf Advanced Code Performance Analysis Through Hierarchical Profiling. http://tracepoint.galatia.com/noframes/products/hiprof/profiling/overview.

[11] 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.

[12] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[13] 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.

[14] Jonathan C. Hardwick and Jay Sipelstein. "Java as an Intermediate Language." Technical Report CMU-CS-96-161. Department of Computer Science. Carnegie Mellon University, August 1996.

[15] 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.

[16] Scott Hudson. Java Based Constructor of Useful Parsers (CUP). http://www.cc.gatech.edu/gvu/people/Faculty/hudson/java_cup/home.html.

[17] Intermetrics. AppletMagic: Ada for Java Virtual Machine. http://www.appletmagic.com.

[18] 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.

[19] 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.

[20] Han B. Lee. *BIT: Bytecode Instrumenting Tool*. MS thesis, University of Colorado, Boulder, CO, July 1997.

[21] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[22] MIPS Computer Systems, Inc. *Assembly Language Programmer's Guide*, 1986.

[23] J. Eliot B. Moss and Antony L. Hosking. "Approaches to Adding Persistence to Java." *First International Workshop on Persistence and Java*, Septermber 1996.

[24] Martin Odersky, Michael Philippsen, and Christian Kemper. EspressoGrinder. http://wwwipd.ira.uka.de/~espresso/.

[25] ODI. *PSE/PSE Pro for Java API User Guide*. 1997.

[26] Srinivasan Parthasarathy, Michael Cierniak, and Wei Li. "NetProf: Network-based High-level Profiling of Java Bytecode." Technical Report 622, Computer Science Department, University of Rochester, May 1996.

[27] 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.

[28] Michael D. Smith. "Tracing with Pixie." Memo from Center for Integrated Systems, Stanford Univ., April 1991.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] Benjamin Zorn and Paul Hilfinger. "A Memory Allocation Profiler for C and Lisp Programs." *USENIX Conference Proceedings*, pages 223-237, Summer 1988.