SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2007; **37**:581–641
Published online 24 October 2006 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.776

An empirical study of Java bytecode programs

Christian Collberg, Ginger Myles*,† and Michael Stepp

Department of Computer Science, University of Arizona, Tucson, AZ 85721, U.S.A.

SUMMARY

We present a study of the static structure of real Java bytecode programs. A total of 1132 Java jar-files were collected from the Internet and analyzed. In addition to simple counts (number of methods per class, number of bytecode instructions per method, etc.), structural metrics such as the complexity of control-flow and inheritance graphs were computed. We believe this study will be valuable in the design of future programming languages and virtual machine instruction sets, as well as in the efficient implementation of compilers and other language processors. Copyright © 2006 John Wiley & Sons, Ltd.

Received 18 August 2004; Revised 28 June 2006; Accepted 28 June 2006

KEY WORDS: Java; bytecode; measure; software complexity metrics

1. INTRODUCTION

In a much cited study [1], Knuth examined FORTRAN programs collected from printouts found in a computing center. Among other things, he found that arithmetic expressions tend to be small, which, he argued, has consequences for code-generation and optimization algorithms chosen in a compiler. Similar studies have been carried out for COBOL [2,3], Pascal [4], and APL [5,6] source code.

In this paper we report on a study on the static structure of real Java bytecode programs. Using information gathered from an automated Google search, we collected a sample of 1132 Java programs, in the form of jar-files (collections of Java class files). The static structure of these programs was analyzed automatically using SandMark, a tool which, among other things, performs static analysis of Java bytecode.

It is our hope that the information gathered and presented here will be of use in a variety of settings. For example, information about the structure of real programs in one language can be used to design

*Correspondence to: Ginger Myles, Department of Computer Science, University of Arizona, Tucson, AZ 85721, U.S.A.
†E-mail: gmyles@gmail.com

Contract/grant sponsor: National Science Foundation; contract/grant number: 324360
Contract/grant sponsor: Air Force Research Lab; contract/grant number: F33615-02-C-1146

future languages in the same family. One example is the `finally` clause of Java exception handlers. Special instructions (`jsr` and `ret`) were added to Java bytecode to handle this construct efficiently. These instructions turn out to be a major source of complexity for the Java verifier [7]. If, instead, the Java bytecode designers had known (from a study of MODULA-3 programs, for example) that the `finally` clause is very unusual in real programs, they may have elected to keep `jsr`/`ret` out of the instruction set. This would have simplified the Java bytecode verifier while imposing little overhead on typical programs[‡].

There are many types of tools that operate on programs. Compilers are an obvious example, but there are many software engineering tools which transform programs in order to improve on their structure, readability, modifiability, etc. Such language processors can benefit from knowing typical and extreme counts of various aspects of real programs. For example, in our study we have found that while, on average, a Java class file has 9.0 methods, in the extreme case we found a class with 570 methods. This information can be used to select appropriate data structures, algorithms, and memory allocation strategies.

As a further example, we have found that the average Java class has no more than one method that overrides a method of its superclass. This means that most methods are written 'from scratch', and will not be present in any of the superclasses of that class. Furthermore, it means that most methods written in a given class are unlikely to be overridden in its subclasses. Thus, aggressive inlining appears to be a good candidate for optimization. The usual obstacle to inlining in Java is virtual method invocation, where a single method callsite could have many potential targets. However, given these data, we can see that often this will not be a problem, because methods are rarely overridden. Combined with the fact that the average method has 33.2 instructions and is thus quite small, we see that aggressive inlining is an excellent candidate for optimization. We hope that this study will be useful in providing many other insights that will facilitate the design of tools to study and improve programs.

Our own research is focused on the protection of software from piracy, tampering, and reverse engineering, using code obfuscation and software watermarking [8]. Code obfuscation attempts to introduce confusion in a program to slow down an adversary's attempts at reverse engineering it. Software watermarking inserts a copyright notice or customer identification number into a program to allow the owner to assert their intellectual property rights. An important aspect of these techniques is *stealth*. For example, a software watermarking algorithm should not embed a mark by inserting code that is highly unusual, since that would make it easy to locate and remove. Our study of instruction frequencies and instruction n-grams directly addresses this concern, by showing us exactly which instruction sequences are common and which are not. For example, any code that contains the `JSR_W` or `GOTO_W` instructions would be extremely unstealthy, since not a single one of our 1132 test jars contain either of these instructions. We believe the information presented in this paper will be useful in developing future evaluation models for the stealth of software protection algorithms.

The remainder of this paper is structured as follows. In Section 2 we describe how our statistics were gathered. In Section 3 we give a brief overview of Java bytecode. In Sections 4, 5, 6, and 7, we present application-level, class-level, method-level, and instruction-level statistics, respectively. In Section 8 we discuss related work, and in Section 9 we summarize our findings.

[‡]The `finally` clause can be implemented by copying code.

DOI: 10.1002/spe

Table I. Collected jar-file statistics.

Measure	Count
Total number of jar-files	1132
Total size of jar-files (bytes)	198 945 317
Total number of class files	102 688
Total number of packages	7682
Total number of classes	90 500
Total number of interfaces	12 188
Total number of constant pool entries	12 538 316
Total number of methods	874 115
Total number of fields	422 491
Total number of instructions	26 597 868

2. EXPERIMENTAL METHODOLOGY

Table I shows some statistics of the applications that were gathered. Figure 1 shows an overview of how our statistics were collected.

To obtain a suitably random set of sample data, we queried the Google search engine using the key-phrase '`"index of" jar`'. This query was designed to find Web pages that display server directory listings that contain files with the extension `.jar`. In the resulting HTML pages we searched for any `<A>` tag whose `HREF` attribute designated a jar-file. These files were then downloaded.

The initial collection of jar-files numbered in excess of 2000. An initial analysis discarded any files that contained no Java classes, or were structurally invalid. Static statistics were next gathered using the SandMark tool.

SandMark [9] is a tool developed to aid in the study of software-based software protection techniques. The tool is implemented in Java and operates on Java bytecode. Included in SandMark are algorithms for code obfuscation, software watermarking, and software birthmarking. A variety of static analysis techniques are included to aid in the development of new algorithms and as a means to study the effectiveness of these algorithms. Examples of such techniques are: class hierarchy, control-flow, and call graphs; def-use and liveness analysis; stack simulation; forward and backward slicing; various bytecode diffing algorithms; a bytecode viewer; and a variety of software complexity metrics.

Not all well-formed jar-files could be completely analyzed. In most cases this was because the jar-file was not self-contained, i.e. it referenced classes that were not in the jar or in the Java standard library. Missing class files prevent the class hierarchy from being constructed, for example. In these cases we still computed as many statistics as possible. For example, while an incomplete class hierarchy prevented us from gathering accurate statistics of class inheritance depth, it still allowed us to gather control-flow graph (CFG) statistics. Our SandMark tool is also not perfect. In particular, it is known to build erroneous CFGs for methods with complex subroutine structures (combinations of the `jsr` and `ret` instructions used for Java's `finally` clause). There are few such CFGs in our sample set, so this problem is unlikely to adversely affect our data.

Owing to our random sampling of jar-files from the Internet, the collection is somewhat idiosyncratic. We assume that any two jar-files with the same name are in fact the same program, and

DOI: 10.1002/spe

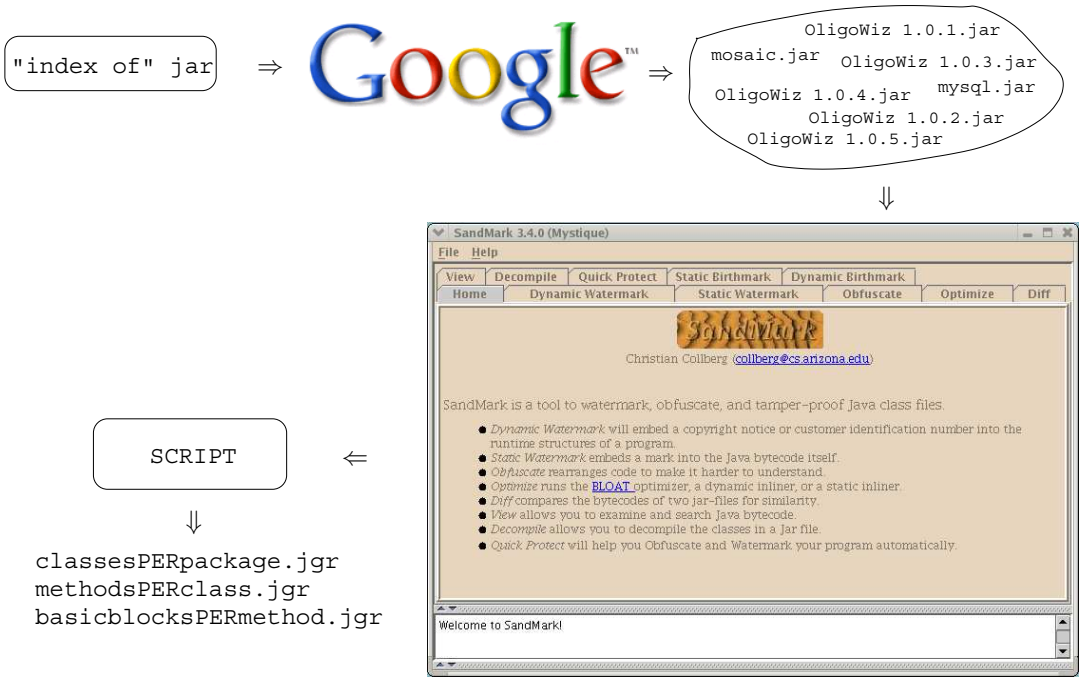


Figure 1. Overview of how our statistics were gathered.

keep only one. However, we kept those files whose names indicated that they were different versions of the same program, as shown by the `OligoWiz` files in Figure 1. Most likely, these files are very similar and may contain methods that are identical between versions. It is reasonable to assume that such redundancy will have somewhat skewed our results. An alternative strategy might have been to guess (based on the file name) which files are versions of the same program, and keep only the higher-numbered file. A less random sampling of programs could also have been collected from well-known repositories of Java code, such as `sourceforge.net`.

Giving an informative presentation of this type of data turns out to be difficult. In many applications we will only be interested in *typical* values (such as *mode* or *mean*) or extreme values (such as *min* and *max*). Such values can easily be presented in tabular form. However, we would also like to be able to quickly get a general 'feel' for the behavior of the data, and this is best presented in a visual form. The visualization is complicated by the fact that most of our data have sharp 'spikes' and long 'tails'. That is, one or a few (typically small) values are very common, but there are a small number of large outliers which by themselves are also interesting. This can be seen, for example, in Figure 30(b) below, which shows that out of the 801 117 methods in our data, 99% have fewer than two subroutines but one method has 29 subroutines.

We have chosen to visualize much of our data using binned bar graphs where extremely tall bars are truncated to allow small values to be visualized. For example, consider the graph in Figure 2 which

DOI: 10.1002/spe

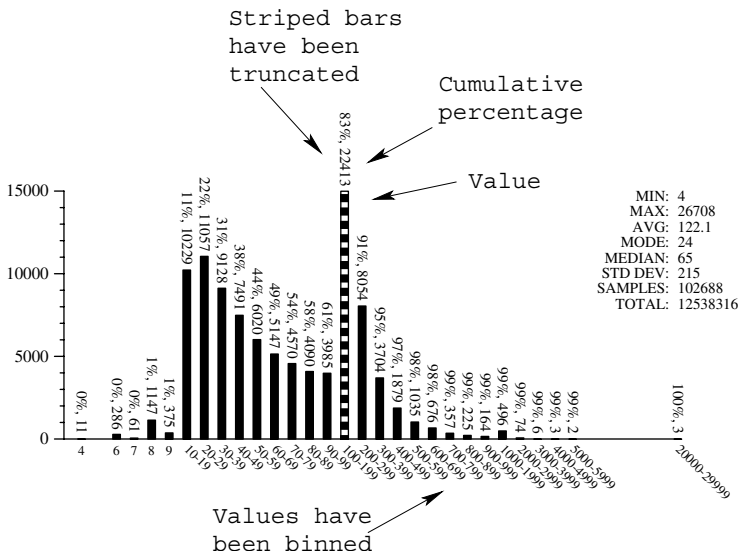


Figure 2. Illustration and explanation of the bar graphs used throughout the paper for data visualization.

shows the number of constants in the constant pool of the Java applications we studied. Most of our graphs will have the same structure. Along the x-axis we show the bins into which our data have been classified. On top of each bar the actual count and cumulative percentage are shown. Very tall bars are truncated and shown striped. In a separate table to the top right of the graph we show the total number of data points, the minimum, maximum, and average x-values, the *mode*[§], the *median* (the middle value), and the standard deviation. The `SAMPLES` value is the total number of items inspected for the given statistic, and the `TOTAL` value is the total number of sub-items counted. For example, in the above graph, the `SAMPLES` value will be the number of classes analyzed and the `TOTAL` value will be the sum of all of the constant pool entries over all of the classes analyzed. The `TOTAL` value is only included where appropriate. The `FAILED` value gives the number of unsuccessful measurements, when appropriate.

3. THE STRUCTURE OF JAVA BYTECODE PROGRAMS

A Java application consists of a collection of classes and interfaces. Each class or interface is compiled into a *class file*. A program consists of a number of class files which are collected together into a *jar-file*.

[§]The mode is the most frequently occurring value. This is often—but because of binning not always—the tallest bar of the graph.

DOI: 10.1002/spe

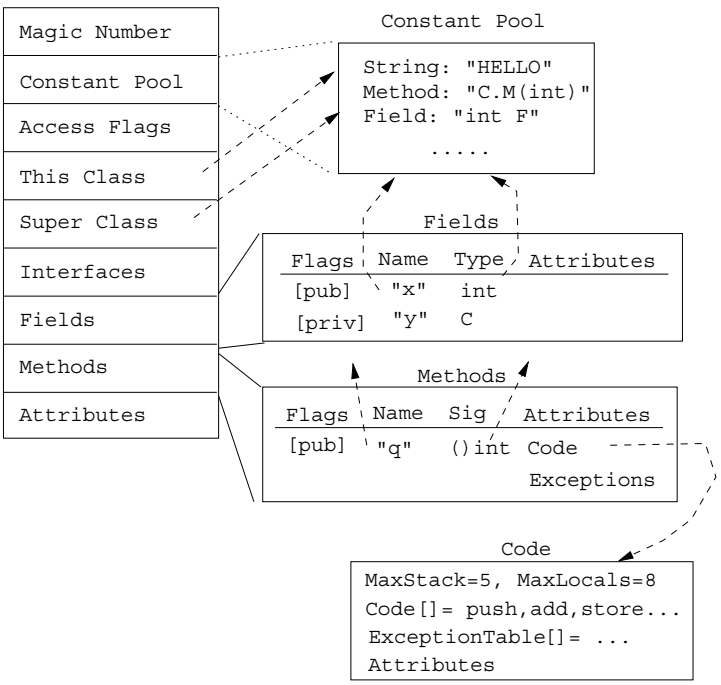


Figure 3. A view of the Java class file format.

A jar-file is directly executable by a Java virtual machine interpreter. The Java class file stores all necessary data regarding the class. There is a symbol table (called the *Constant Pool*) which stores strings, large literal integers and floats, and names and types of all fields and methods. Each method is compiled to Java bytecode, a stack-based virtual machine instruction set. Figure 3 shows the structure of the Java class file format. The JVM is defined by Lindholm and Yellin [10].

The Java bytecodes can manipulate data in several formats: integers (32 bits), longs (64 bits), floats (32 bits), doubles (64 bits), shorts (16 bits), bytes (8 bits), Booleans (1 bit), chars (16 bit Unicode), object references (32 bit pointers), and arrays. The Boolean, byte, char, and short types are compiled down into integers.

Bytecode instructions have variable widths. Simple instructions such as `iadd` (integer addition) are one byte wide, while some instructions (such as `tableswitch`) can be multiple bytes. Each method can have up to 65 536 local variables and formal parameters, called *slots*. The bytecodes reference slots by number. For example, the instruction `'iload_3'` pushes the third local variable onto the stack. In order to access high-numbered slots, a special `wide` instruction can be used to modify load and store instructions to use 16-bit indexes. The Java execution stack is 32 bits wide. Longs and doubles take up two stack entries and two slot numbers.

DOI: 10.1002/spe

Table II. Notation used to refer to data values in the bytecode.

Notation	Explanation
$\mathcal{B}$	An 8-bit integer value
$\mathcal{S}$	A 16-bit integer value
$\mathcal{L}$	A 32-bit integer value
$\mathcal{C}_b$	An 8-bit constant pool index
$\mathcal{C}_s$	A 16-bit constant pool index
$\mathcal{F}_b$	An 8-bit local variable index
$\mathcal{F}_s$	A 16-bit local variable index
$C\,[i\,]$	The ith constant pool entry
$V\,[i\,]$	The ith variable/formal parameter in the current method

Local variable slots are untyped. In fact, a particular slot can hold different types of data at different locations in a method. However, regardless of how execution reaches a given location in the method, the type of data stored in a particular slot at that location will always be the same. A static analysis known as a *stack simulation* can compute slot types without executing a method.

Some bytecodes reference data from the class' constant pool, for example to push large constants or to invoke methods. Constant pool references are 8 or 16 bits long. To push a reference to a literal string with constant pool number 4567, the compiler would issue the instruction `'ldc_w 4567'`. If the constant pool number instead fits into a byte (such as 123), the shorter instruction `'ldc 123'` would suffice.

Some information is stored in *attributes* in the class file. This includes exception table ranges, and (for debugging) line-number ranges and local variable names.

Table II explains the notation used in Tables III–VI, which give an overview of the JVM instruction set.

4. PROGRAM-LEVEL STATISTICS

In this and the following three sections we will present the data collected about applications (this section), classes (Section 5), methods (Section 6), and instructions (Section 7).

Figures 4–7 visualize application-level data about the programs we gathered.

4.1. Packages

Classes in Java are optionally organized into a hierarchy of *packages*. For example, Java's `String` class is in the package `java.lang`, and can be referred to as `java.lang.String`. As can be seen from Figure 4(a), many Java programs put all classes into the same package. In fact, half of the 1132 applications we gathered have three or fewer packages, and only four have 50 or more.

A package α is counted if there exists some class β such that the fully qualified classname of β is $\alpha.\beta$. Thus, if an application has classes *java.pack1.Class1* and *java.pack2.Class2* then *java.pack1* and

DOI: 10.1002/spe

Table III. The first 87 Java bytecode instructions.

Opcode	Mnemonic	Args	Stack	Description
0	`nop`		$[] \Rightarrow []$	
1	`aconst_null`		$[] \Rightarrow$ [null]	Push null object.
2	`iconst_m1`		$[] \Rightarrow [-1]$	Push -1.
3 . . . 8	`iconst_`n		$[] \Rightarrow [n]$	Push integer constant n, $0 \leq n \leq 5$.
9 . . . 10	`lconst_`n		$[] \Rightarrow [n]$	Push long constant n, $0 \leq n \leq 1$.
11 . . . 13	`fconst_`n		$[] \Rightarrow [n]$	Push float constant n, $0 \leq n \leq 2$.
14 . . . 15	`dconst_`n		$[] \Rightarrow [n]$	Push double constant n, $0 \leq n \leq 1$.
16	`bipush`	$n{:}\mathcal{B}$	$[] \Rightarrow [n]$	Push 1-byte signed integer.
17	`sipush`	$n{:}\mathcal{S}$	$[] \Rightarrow [n]$	Push 2-byte signed integer.
18	`ldc`	$n{:}\mathcal{C}_b$	$[] \Rightarrow [C[n]]$	Push item from constant pool.
19	`ldc_w`	$n{:}\mathcal{C}_s$	$[] \Rightarrow [C[n]]$	Push item from constant pool.
20	`ldc2_w`	$n{:}\mathcal{C}_s$	$[] \Rightarrow [C[n]]$	Push long/double from constant pool.
21 . . . 25	X`load`	$n{:}\mathcal{F}_b$	$[] \Rightarrow [V[n]]$	$X \in$ {i,l,f,d,a}, Load int, long, float, double, object from local var.
26 . . . 29	`iload_`n		$[] \Rightarrow [V[n]]$	Load local integer var n, $0 \leq n \leq 3$.
30 . . . 33	`lload_`n		$[] \Rightarrow [V[n]]$	Load local long var n, $0 \leq n \leq 3$.
34 . . . 37	`fload_`n		$[] \Rightarrow [V[n]]$	Load local float var n, $0 \leq n \leq 3$.
38 . . . 41	`dload_`n		$[] \Rightarrow [V[n]]$	Load local double var n, $0 \leq n \leq 3$.
42 . . . 45	`aload_`n		$[] \Rightarrow [V[n]]$	Load local object var n, $0 \leq n \leq 3$.
46 . . . 53	X`load`		$[A, I] \Rightarrow [V]$	$X \in$ {ia,la,fa,da,aa,ba,ca,sa}. Push the value V (an int, long, etc.) stored at index I of array A.
54 . . . 58	X`store`	$n{:}\mathcal{F}_b$	$[V[n]] \Rightarrow []$	$X \in$ {i,l,f,d,a}, Store int, long, float, double, object to local var.
59 . . . 62	`istore_`n		$[V[n]] \Rightarrow []$	Store to local integer var n, $0 \leq n \leq 3$.
63 . . . 66	`lstore_`n		$[V[n]] \Rightarrow []$	Store to local long var n, $0 \leq n \leq 3$.
67 . . . 70	`fstore_`n		$[V[n]] \Rightarrow []$	Store to local float var n, $0 \leq n \leq 3$.
71 . . . 74	`dstore_`n		$[V[n]] \Rightarrow []$	Store to local double var n, $0 \leq n \leq 3$.
75 . . . 78	`astore_`n		$[V[n]] \Rightarrow []$	Store to local object var n, $0 \leq n \leq 3$.
79 . . . 86	X`store`		$[A, I, V] \Rightarrow []$	$X \in$ {ia,la,fa,da,aa,ba,ca,sa}. Store the value V (an int, long, etc.) at index I of array A.

java.pack2 would be counted, but *java* would not. Also, the default or 'null' package is counted exactly once, if there is a class in that package.

Figure 5(a) shows that while a small number of programs have packages with hundreds of classes, the typical package will have only one, and the average is about 11.8.

Packages can be nested inside of other packages, allowing for the easy creation of unique names. While it is possible to create a package hierarchy of arbitrary depth, Figure 4(b) shows that the maximum depth for an application is 8, with an average depth of 3.9.

A Java *interface* is a special type of class that only contains constant declarations or method signatures. A class which *implements* an interface must provide implementations of the methods.

DOI: 10.1002/spe

Table IV. Java bytecode instructions 87 to 169.

Opcode	Mnemonic	Args	Stack	Description
87	pop		$[A] \Rightarrow []$	Pop top of stack.
88	pop2		$[A, B] \Rightarrow []$	Pop 2 elements.
89	dup		$[V] \Rightarrow [V, V]$	Duplicate top of stack.
90	dup_x1		$[B, V] \Rightarrow [V, B, V]$	
91	dup_x2		$[B, C, V] \Rightarrow [V, B, C, V]$	
92	dup2		$[V, W] \Rightarrow [V, W, V, W]$	
93	dup2_x1		$[A, V, W] \Rightarrow [V, W, A, V, W]$	
94	dup2_x2		$[A, B, V, W] \Rightarrow [V, W, A, B, V, W]$	
95	swap		$[A, B] \Rightarrow [B, A]$	Swap top stack elements.
96 . . . 99	Xadd		$[A, B] \Rightarrow [R]$	$X \in${i,l,d,f}. $R = A + B$.
100 . . . 103	Xsub		$[A, B] \Rightarrow [R]$	$X \in${i,l,d,f}. $R = A - B$.
104 . . . 107	Xmul		$[A, B] \Rightarrow [R]$	$X \in${i,l,d,f}. $R = A * B$.
108 . . . 111	Xdiv		$[A, B] \Rightarrow [R]$	$X \in${i,l,d,f}. $R = A/B$.
112 . . . 115	Xrem		$[A, B] \Rightarrow [R]$	$X \in${i,l,d,f}. $R = A\%B$.
116 . . . 119	Xneg		$[A] \Rightarrow [R]$	$X \in${i,l,d,f}. $R = -A$.
120 . . . 121	Xshl		$[A, B] \Rightarrow [R]$	$X \in${i,l}. $R = A << B$.
122 . . . 123	Xshr		$[A, B] \Rightarrow [R]$	$X \in${i,l}. $R = A >> B$.
124 . . . 125	Xushr		$[A, B] \Rightarrow [R]$	$X \in${i,l}. $R = A >>> B$.
126 . . . 127	$X$and		$[A, B] \Rightarrow [R]$	$X \in${i,l}. $R = A\&B$.
128 . . . 129	Xor		$[A, B] \Rightarrow [R]$	$X \in${i,l}. $R = A\|B$.
130 . . . 131	Xxor		$[A, B] \Rightarrow [R]$	$X \in${i,l}. $R = A \text{xor} B$.
132	iinc	$V{:}\mathcal{F}_b, B{:}\mathcal{B}$	$[] \Rightarrow []$	$V += B$.
133 . . . 144	$X2Y$		$[F] \Rightarrow [T]$	Convert F from type X to T of type Y. $X \in${i,l,f,d}, $Y \in${i,l,f,d}.
145 . . . 147	i2X		$[F] \Rightarrow [T]$	$X \in${b,c,s}. Convert integer F to byte, char, or short.
148	lcmp		$[A, B] \Rightarrow [V]$	Compare long values. $A > B \Rightarrow V = 1$, $A < B \Rightarrow V = -1$, $A = B \Rightarrow V = 0$.
149,151	Xcmpl		$[A, B] \Rightarrow [V]$	Compare float or double values. $X \in${f,d}. $A > B \Rightarrow V = 1$, $A < B \Rightarrow V = -1$, $A = B \Rightarrow V = 0$. $A = \text{NaN} \vee B = \text{NaN} \Rightarrow V = -1$
150,152	Xcmpg		$[A, B] \Rightarrow [V]$	Compare float or double values. $X \in${f,d}. $A > B \Rightarrow V = 1$, $A < B \Rightarrow V = -1$, $A = B \Rightarrow V = 0$. $A = \text{NaN} \vee B = \text{NaN} \Rightarrow V = 1$
153 . . . 158	if$\diamond$	$L{:}\mathcal{S}$	$[A] \Rightarrow []$	$\diamond$={eq,ne,lt,ge,gt,le}. If $A \diamond 0$ goto $L + \text{pc}$.
159 . . . 164	if_icmp$\diamond$	$L{:}\mathcal{S}$	$[A, B] \Rightarrow []$	$\diamond$={eq,ne,lt,ge,gt,le}. If $A \diamond B$ goto $L + \text{pc}$.
165 . . . 166	if_acmp$\diamond$	$L{:}\mathcal{S}$	$[A, B] \Rightarrow []$	$\diamond$={eq,ne}. A, B are object refs. If $A \diamond B$ goto $L + \text{pc}$.
167	goto	$I{:}\mathcal{S}$	$[] \Rightarrow []$	Jump to $I + \text{pc}$.
168	jsr	$I{:}\mathcal{S}$	$[] \Rightarrow [A]$	Jump subroutine to instruction $I + \text{pc}$. A = the address of the instruction after the jsr.
169	ret	$L{:}\mathcal{F}_b$	$[] \Rightarrow []$	Return from subroutine. Address in local var L.

Interfaces are often used to compensate for Java's lack of multiple inheritance. Figure 5(b) shows that over 70% of Java packages contain 0 or 1 interface.

4.2. Protection

A Java class can be declared as *abstract* (it serves only as a superclass to classes which actually implements its methods) or *final* (it cannot be extended). These declarations are, however, fairly unusual. Figures 6(a) and 6(b) show that over 70% of all packages contain no abstract or final classes.

DOI: 10.1002/spe

Table V. Java bytecode instructions 170 to 195.

Opcode	Mnemonic	Args	Stack
170	`tableswitch`	D:$\mathcal{L}$,l, h:$\mathcal{L}$,o^{h-l+1}	$[K] \Rightarrow []$
	JuLongDescrmp through the K:th offset. Else goto D.		
171	`lookupswitch`	D:$\mathcal{L}$,n:$\mathcal{L}$,$(m, o)^n$	$[K] \Rightarrow []$
	If, for one of the (m, o) pairs, $K = m$, then goto o. Else goto D.		
172 . . . 176	X`return`		$[V] \Rightarrow []$
	$X \in${i,f,l,d,a}. Return V.		
177	`return`		$[] \Rightarrow []$
	Return from void method.		
178	`getstatic`	F:$\mathcal{C}_s$	$[] \Rightarrow [V]$
	Push value V of static field F.		
179	`putstatic`	F:$\mathcal{C}_s$	$[V] \Rightarrow []$
	Store value V into static field F.		
180	`getfield`	F:$\mathcal{C}_s$	$[R] \Rightarrow [V]$
	Push value V of field F in object R.		
181	`putfield`	F:$\mathcal{C}_s$	$[R, V] \Rightarrow []$
	Store value V into field F of object R.		
182	`invokevirtual`	P:$\mathcal{C}_s$	$[R, A_1, A_2, \ldots] \Rightarrow []$
	Call virtual method P, with arguments $A_1 \cdots A_n$, through object reference R.		
183	`invokespecial`	P:$\mathcal{C}_s$	$[R, A_1, A_2, \ldots] \Rightarrow []$
	Call private/init/superclass method P, with arguments $A_1 \cdots A_n$, through object reference R.		
184	`invokestatic`	P:$\mathcal{C}_s$	$[A_1, A_2, \ldots] \Rightarrow []$
	Call static method P with arguments $A_1 \cdots A_n$.		
185	`invokeinterface`	P:$\mathcal{C}_s$,n:$\mathcal{S}$	$[R, A_1, A_2, \ldots] \Rightarrow []$
	Call interface method P, with n arguments $A_1 \cdots A_n$, through object reference R.		
187	`new`	T:$\mathcal{C}_s$	$[] \Rightarrow [R]$
	Create a new object R of type T.		
188	`newarray`	T:$\mathcal{B}$	$[C] \Rightarrow [R]$
	Allocate new array R, element type T, C elements long.		
189	`anewarray`	T:$\mathcal{C}_s$	$[C] \Rightarrow [A]$
	Allocate new array A of reference types, element type T, C elements long.		
190	`arraylength`		$[A] \Rightarrow [L]$
	Determines the length L of array A.		
191	`athrow`		$[R] \Rightarrow [?]$
	Throw exception.		
192	`checkcast`	C:$\mathcal{C}_s$	$[R] \Rightarrow [R]$
	Ensures that R is of type C.		
193	`instanceof`	C:$\mathcal{C}_s$	$[R] \Rightarrow [V]$
	Push 1 if object R is an instance of class C, else push 0.		
194	`monitorenter`		$[R] \Rightarrow []$
	Get lock for object R.		
195	`monitorexit`		$[R] \Rightarrow []$
	Release lock for object R.		

DOI: 10.1002/spe

Table VI. Java bytecode instructions 196 to 201.

Opcode	Mnemonic	Args	Stack
196	`wide`	$C{:}\mathcal{B}, I{:}\mathcal{F}_s$	$[] \Rightarrow []$
	Perform opcode C on variable V `[`I`]`. C is one of the load/store instructions.		
197	`multianewarray`	$T{:}\mathcal{C}_s, D{:}\mathcal{C}_b$	$[d_1, d_2, \ldots] \Rightarrow [R]$
	Create new D-dimensional multidimensional array R. $d_1, d_2, \ldots$ are the dimension sizes.		
198	`ifnull`	$L{:}\mathcal{S}$	$[V] \Rightarrow []$
	If $V =$ `null` goto L.		
199	`ifnonnull`	$L{:}\mathcal{S}$	$[V] \Rightarrow []$
	If $V \neq$ `null` goto L.		
200	`goto_w`	$I{:}\mathcal{L}$	$[] \Rightarrow []$
	Goto instruction I.		
201	`jsr_w`	$I{:}\mathcal{L}$	$[] \Rightarrow [A]$
	Jump subroutine to instruction I. A is the address of the instruction right after the jsr_w.		

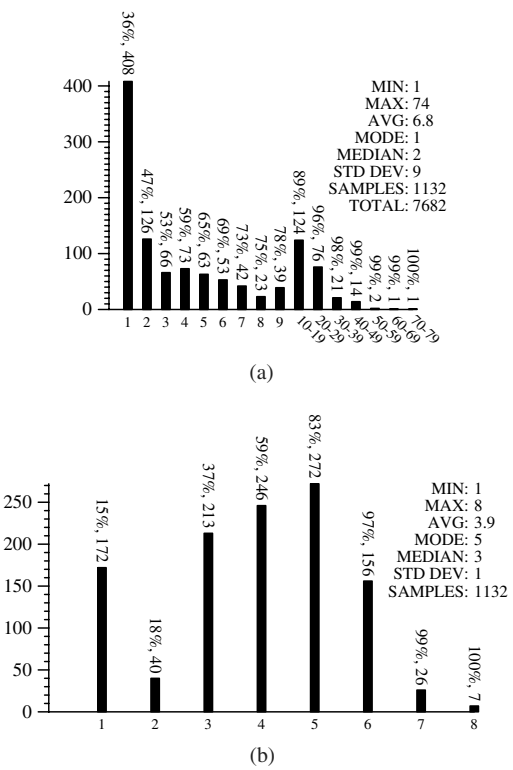


Figure 4. Program-level statistics: (a) number of packages per application; (b) package depth per application.

DOI: 10.1002/spe

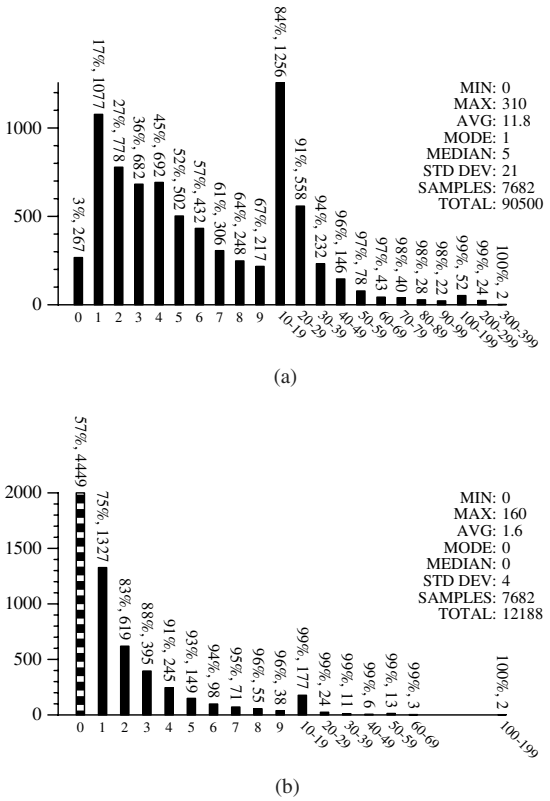


Figure 5. Program-level statistics: (a) number of classes per package; (b) number of interfaces per package.

4.3. Inheritance graphs

In addition to a class implementing an interface, a class can also extend another class. In this case the subclass inherits all of the variables and methods of the class which it extends (the superclass), thus creating an inheritance relationship. An *inheritance graph* can be constructed to represent the superclass/subclass relationship. An inheritance graph is a rooted, directed, acyclic graph where the nodes are classes and interfaces. There is an edge from node *A* to node *B* iff node *B* directly extends or implements node *A*. Thus, classes will have edges to their direct subclasses, and interfaces will have edges to the classes that implement them and to the interfaces that extend them. In order to make this a rooted graph, we assert that all interfaces extend the class `java.lang.Object` (and, hence, `java.lang.Object` becomes the root). Using this definition, we can see that node *B* is reachable from node *A* iff it is possible to assign a value of type *B* to a variable of type *A*. The *inheritance graph*

DOI: 10.1002/spe

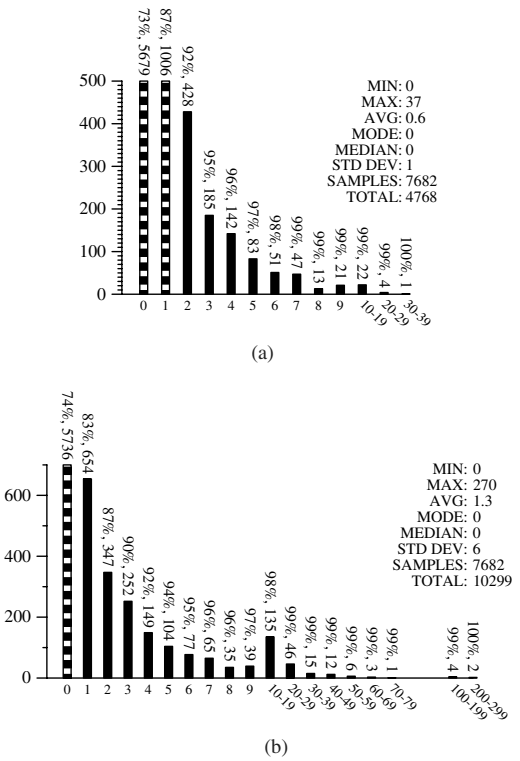


Figure 6. Program-level statistics: (a) number of abstract classes per package; (b) number of final classes package.

height for a given application is the maximum number of superclasses that any class in the application has. This will include some but not all of the Java library classes.

Figure 7(a) shows that on average the height of an inheritance graph for an application is 4.5 and that over 90% of all applications have an inheritance graph with a height of less than 7.

It is important to note that for 303 of our applications, the inheritance graph construction failed due to the jar-file not being self-contained. This means that some class in the application had a superclass that was unavailable for analysis, therefore we could not place it correctly in the inheritance graph. This is typical of applications that rely on external libraries which are not packaged with them.

Figure 7(b) shows the number of classes in each application that extend other classes in the same application, as opposed to Java library classes. Some classes in each application *must* directly extend a Java library class (most often `java.lang.Object`), but it is interesting to note that only about one-third (32 899/90 500) extend other classes inside the same application. Table VII shows that most of the classes in each application extend `java.lang.Object` directly.

DOI: 10.1002/spe

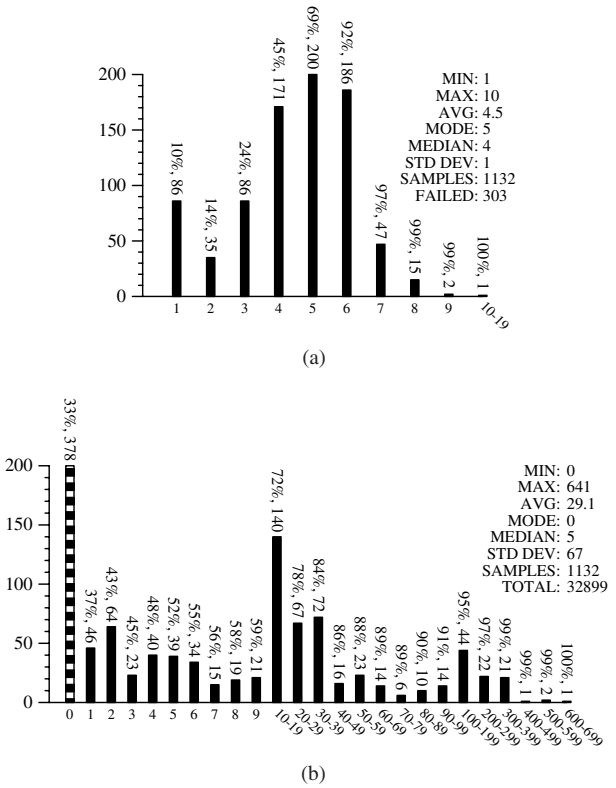


Figure 7. Inheritance graphs: (a) height per application; (b) number of user-class extenders per application.

5. CLASS-LEVEL STATISTICS

In this section we present data regarding the top-level structure of class files. This includes the number, type/signature, and protection of fields and methods, and the class' or interface's position in the application's inheritance graph.

5.1. Fields

A Java class can contain data members, called *fields*. Fields are either *class variables* (they are declared `static` and only one instance exists at runtime) or *instance variables* (every instantiation of the class contains a unique copy).

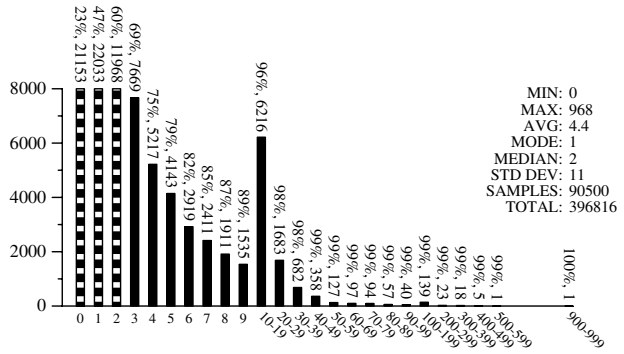
Figures 8–10 show field statistics. In Figure 8(a) we see that 60% of all classes have two or fewer fields, but in one extreme case a class declared almost a thousand fields. Instance variables are more common than class variables. On average, a class will contain 2.8 instance variables and 1.6 class variables, and 44% of all classes have more than one instance variable but only 17% have more than

DOI: 10.1002/spe

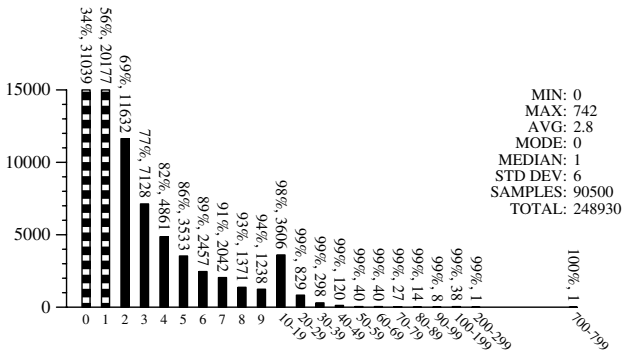
Table VII. Most common standard classes to be extended by application classes.

Class	Count	%
`java.lang.Object`	42 629	47.1
user_class	34 805	38.5
`java.lang.Exception`	1089	1.2
`javax.swing.AbstractAction`	893	1.0
`java.lang.Thread`	738	0.8
`javax.swing.JPanel`	691	0.8
`java.lang.RuntimeException`	464	0.5
`java.awt.event.WindowAdapter`	341	0.4
`java.awt.Panel`	313	0.3
`java.awt.event.MouseAdapter`	309	0.3
`java.util.ListResourceBundle`	276	0.3
`java.util.EventObject`	248	0.3
`java.io.FilterInputStream`	232	0.3
`org.omg.CORBA.portable.ObjectImpl`	226	0.2
`org.omg.CORBA.SystemException`	217	0.2
`org.xml.sax.helpers.DefaultHandler`	203	0.2
`java.awt.Dialog`	203	0.2
`java.io.FilterOutputStream`	202	0.2
`java.applet.Applet`	202	0.2
`java.awt.Canvas`	197	0.2
`java.io.OutputStream`	196	0.2
`java.awt.Frame`	194	0.2
`java.io.IOException`	192	0.2
`java.io.InputStream`	183	0.2
`javax.swing.JFrame`	149	0.2
`javax.swing.JDialog`	135	0.1
`org.omg.CORBA.UserException`	126	0.1
`java.lang.Error`	120	0.1
`java.beans.SimpleBeanInfo`	119	0.1
`java.awt.event.KeyAdapter`	118	0.1
`javax.swing.table.AbstractTableModel`	104	0.1
`java.awt.event.FocusAdapter`	101	0.1
`java.util.AbstractSet`	94	0.1
`java.security.Signature`	80	0.1
`javax.swing.beaninfo.SwingBeanInfo`	79	0.1
`java.security.GeneralSecurityException`	78	0.1
`org.xml.sax.SAXException`	70	0.1
`javax.swing.JComponent`	70	0.1
`javax.swing.event.InternalFrameAdapter`	60	0.1
`java.util.Hashtable`	57	0.1
`java.lang.IllegalArgumentException`	56	0.1
`java.io.Writer`	54	0.1
`java.util.AbstractList`	51	0.1
`java.util.Properties`	50	0.1
`java.io.Reader`	49	0.1

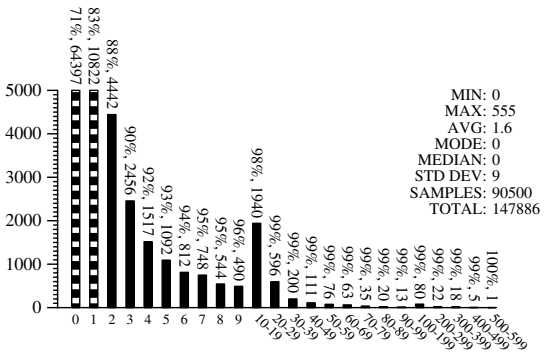
DOI: 10.1002/spe



(a)



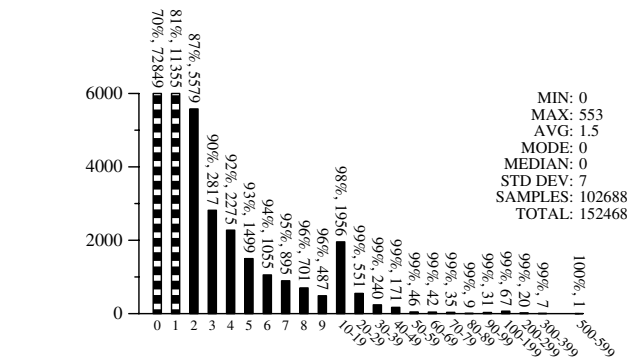
(b)



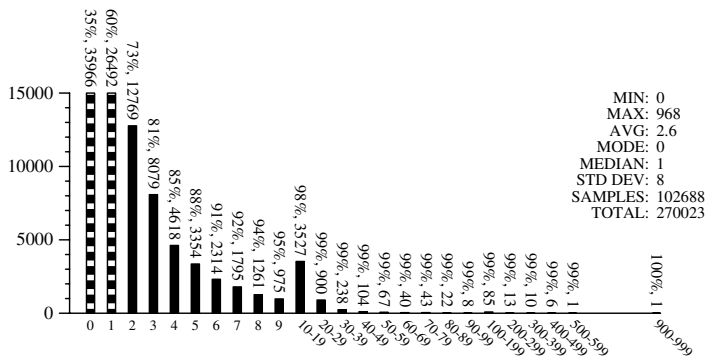
(c)

Figure 8. Field declarations in classes: (a) number of fields per class; (b) number of instance variables per class; (c) number of class (static) variables per class.

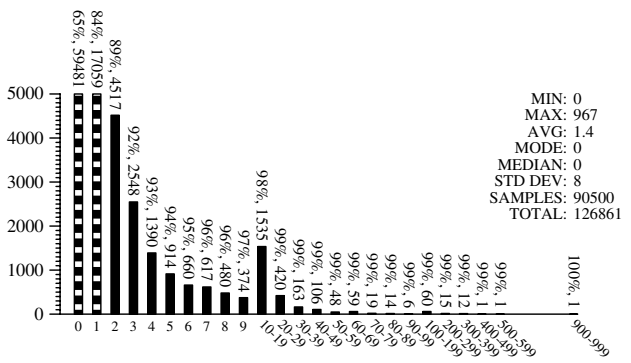
DOI: 10.1002/spe



(a)



(b)



(c)

Figure 9. Field declarations in classes: (a) number of primitive fields per class or interface; (b) number of reference fields per class or interface; (c) number of final fields per class.

DOI: 10.1002/spe

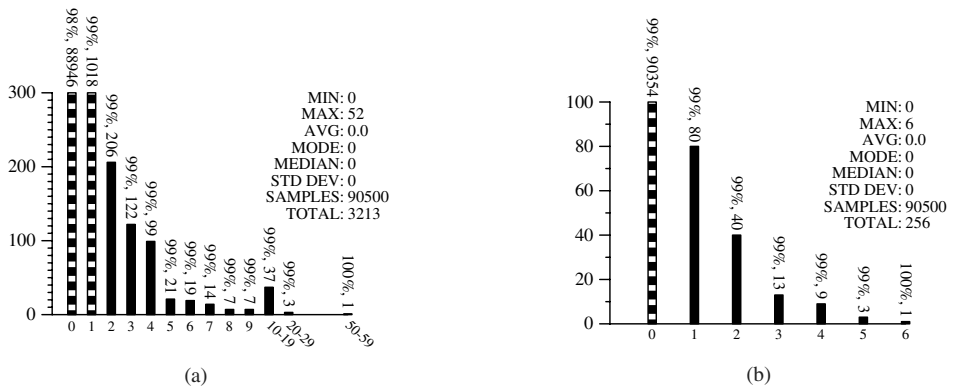


Figure 10. Field declarations in classes: (a) number of transient fields per class; (b) number of volatile fields per class.

one static variable. It is also more common for a class to have fields of reference types rather than primitive types. On average, a class will have 1.5 fields of primitive type, but 2.6 fields of reference type.

Fields may also be declared *final, transient, or volatile*. A final field is one whose value cannot be altered after it is first assigned in the instance or class initializer. A transient field is one that is not part of the persistent state of its parent object. A volatile field is one that cannot be internally cached by the JVM, since it is assumed to be accessed by multiple threads. Figures 9(c), 10(a), and 10(b) report on the number of final, transient, and volatile fields per class, respectively. We see that 98% of all classes have no transient fields, 99% have no volatile fields, and more than half of all classes have no final fields. The rareness of these modifiers makes the outliers in these graphs particularly interesting. While in general there are very few transient fields, Figure 10(a) shows us that one class had 52 of them. Also, when we compare Figures 9(b) and 9(c), we see that they have very similar MAX values. On closer inspection, this is due to a single class in the file '`kawa-1.7.jar`' which has 968 fields, all of which are reference types, and only one of which is not final.

Table VIII gives a breakdown of the declared types of fields. Only primitive types and types exported from the Java standard library are shown. Our data also contained some user defined types with high usage counts. This is due to idiosyncrasies of our collected programs, such as a program declaring vast numbers of fields of one of its classes. Table VIII shows that the vast majority of types are `int`s, `String`s, and `boolean`s. We note that, somewhat surprisingly, `java.lang.Class` (Java's notion of a class) is a frequent field type, and `double`s are more frequent than `float`s.

5.2. Constant pool

Figure 11 shows the number of entries in the *constant pool* (the class file's symbol table) per class. While small literal integers are stored directly in the bytecode, large integers as well as `String`s and real numbers are, instead, stored in the constant pool. Figures 12–14 show the relative distribution of literal types.

DOI: 10.1002/spe

Table VIII. Most common field types.

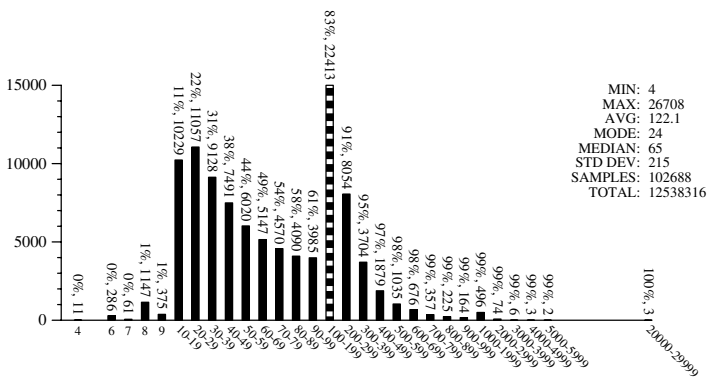
Field type	Count	%
`int`	153 861	21.8
`java.lang.String`	105 787	15.0
`boolean`	44 914	6.4
`java.lang.Class`	24 355	3.4
`long`	16 556	2.3
`java.lang.Object`	14 472	2.0
`byte[]`	10 229	1.4
`int[]`	8157	1.1
`java.util.Vector`	7601	1.0
`java.util.Hashtable`	7095	1.0
`short`	7048	1.0
`byte`	6464	0.9
`java.lang.String[]`	6412	0.9
`java.util.Map`	5692	0.8
`double`	5256	0.7
`java.util.List[]`	4971	0.7
`float`	3115	0.4
`java.io.File`	3019	0.4
`char[]`	2995	0.4
`java.math.BigInteger`	2782	0.3
`java.lang.StringBuffer`	2472	0.3
`java.sql.Connection`	2443	0.3
`javax.swing.JLabel`	2066	0.3
`java.util.HashMap`	2064	0.3
`java.awt.Color`	2058	0.3
`char`	1987	0.3
`java.util.ArrayList`	1748	0.2

The difference between Figures 14(a) and 14(b) is that 'string constants' are user-defined strings, such as all literal strings that appear in the source code. The 'UTF8 strings' include all user strings, but also include strings used internally by the classfile format, as well as the names of all referenced classes, interfaces, methods, fields, etc. It is interesting to note that UTF8 strings comprise over half of the total constants counted, whereas the sum of all numeric constants is less than 2% of the total.

5.3. Methods

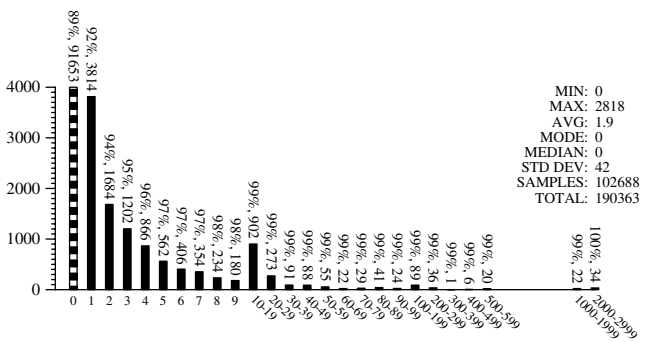
Figures 15–17 give statistics of methods. Of interest is that 73% of all classes have nine or fewer methods (Figure 15(a)), and that the vast majority of classes have no abstract or native methods (Figures 15(b) and 16(a)). Almost all classes have at least one virtual method, with an average of 7.7 methods per class (Figure 17(a)). Static methods are quite rare: 80% of all classes have at most one static method, with an average of 1.3 methods per class (Figure 16(b)).

DOI: 10.1002/spe

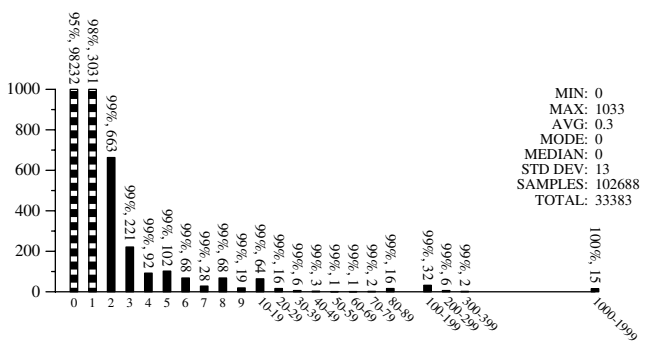


(a)

Figure 11. Number of constant pool entries per class or interface.



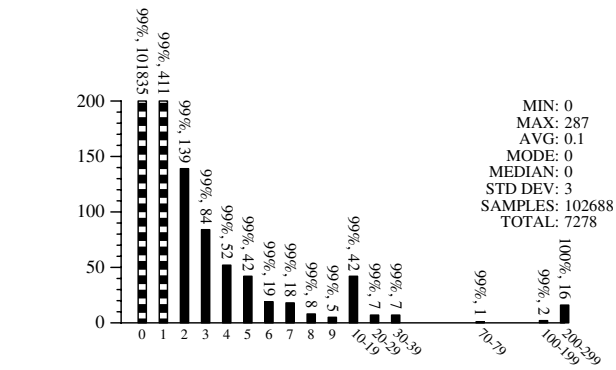
(a)



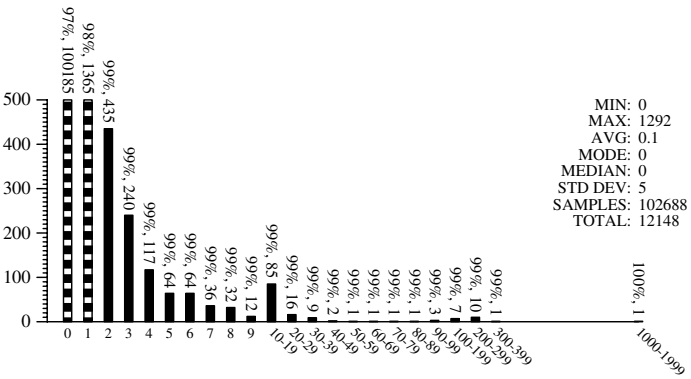
(b)

Figure 12. Literal constants in classes: (a) number of integer entries per class or interface; (b) number of long entries per class or interface.

DOI: 10.1002/spe



(a)



(b)

Figure 13. Literal constants in classes: (a) number of float entries per class or interface; (b) number of double entries per class or interface.

5.4. Member protection

Figures 18 and 19 show the frequency of visibility restrictions of class members (fields and methods). A member can be `package private`, `private`, `protected`, or `public`. Figure 19(c) summarizes the information by giving average numbers of members with a particular protection.

5.5. Inheritance

Figure 20 shows information about class inheritance. Figure 20(a) shows the number of immediate subclasses of a class, i.e. the number of classes that directly extend a particular class. Figure 20(b) shows the number of classes that directly or indirectly extend a particular class. We found that 97% of all classes have two or fewer direct subclasses. One of the classes in our collection is extended

DOI: 10.1002/spe

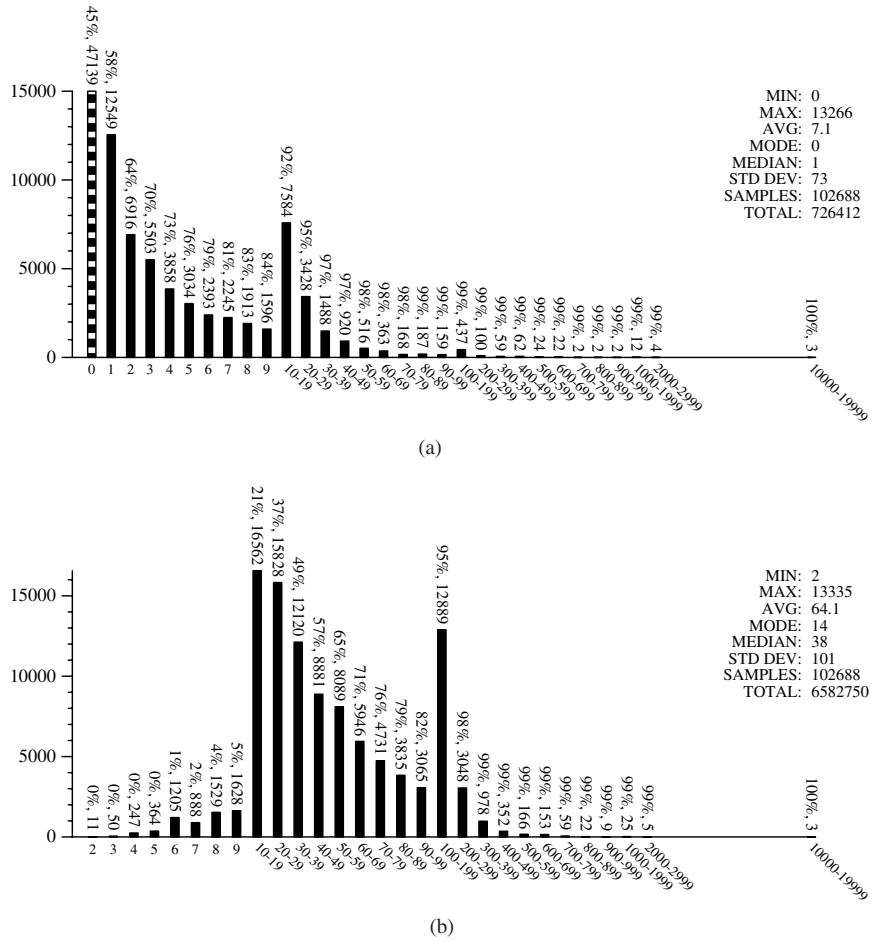


Figure 14. Literal constants in classes: (a) number of string entries per class or interface; (b) number of UTF8 string constants per class.

by 187 classes. In addition, 48% of classes are at depth 1 in the inheritance graph, i.e. they extend `java.lang.Object`, the root of the inheritance graph (Figure 20(c)). The average depth of a class is low (only 2.1), although six of our classes are at depth 30–39. In many cases we failed to build the inheritance hierarchy due to the program containing references to classes not in the jar-file or the standard Java library.

Figure 21 shows the same information for interfaces.

Table VII was computed by looking at which classes each application class extended. Every interface is considered to extend `java.lang.Object`. Similarly, Table IX looks at which interfaces were

DOI: 10.1002/spe

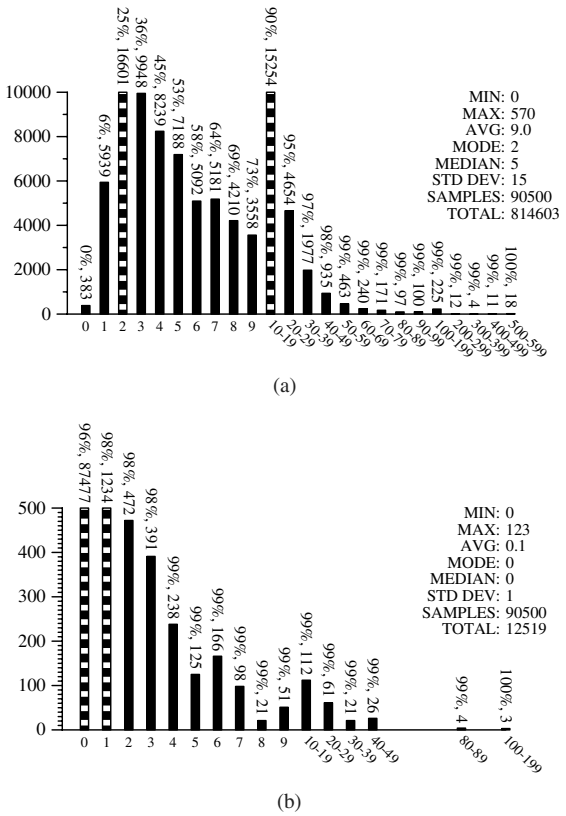


Figure 15. Method declarations in classes: (a) number of methods per class; (b) number of abstract methods per class.

extended by other interfaces. There is a bit of ambiguity here, because in Java source code an interface uses the *extends* keyword to extend another interface, although technically the interface is really being *implemented*. Table X shows which interfaces were implemented by *any* application class, including other interfaces.

Method overriding occurs when a method in a class has the same name and signature as a method in its superclass. This is a technique used to provide a more specialized implementation of a particular method. Figure 22 shows that the majority of classes have at most one overridden method.

6. METHOD-LEVEL STATISTICS

In this section we present method-level statistics. This includes information about method signatures, local variables, CFGs, and exception handlers.

DOI: 10.1002/spe

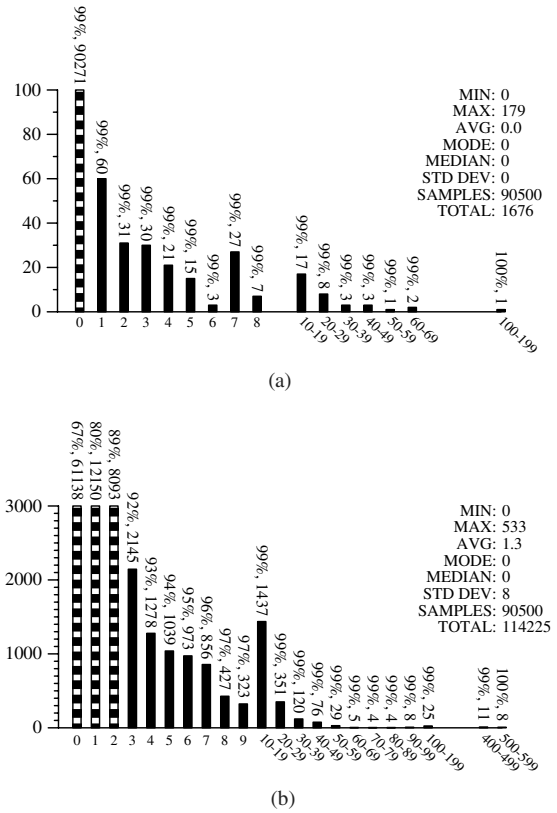


Figure 16. Method declarations in classes: (a) number of native methods per class; (b) number of static methods per class.

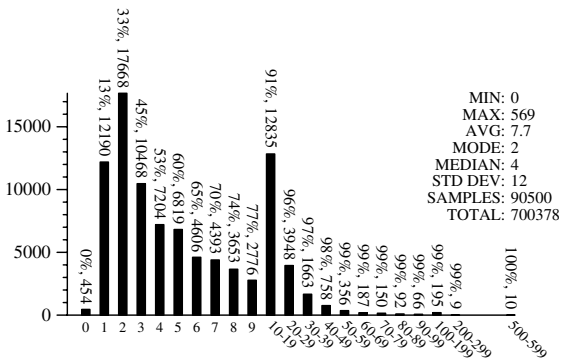
6.1. Method sizes

Figure 23 shows the sizes in bytes and instructions of bytecode methods. The maximum size allowed by the JVM is 65 535 bytes, but only one of our methods (63 019 bytes long) approached this limit.

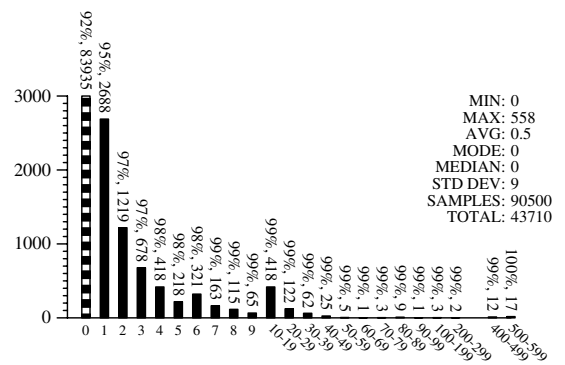
6.2. Local variables and formal parameters

Figure 24 shows the maximum number of slots used by a method. All instance methods will use at least one slot (for the `this` parameter). No method used more than 157 slots, indicating that the `wide` instruction (used to access up to 65 536 slots) will be rarely used.

DOI: 10.1002/spe



(a)



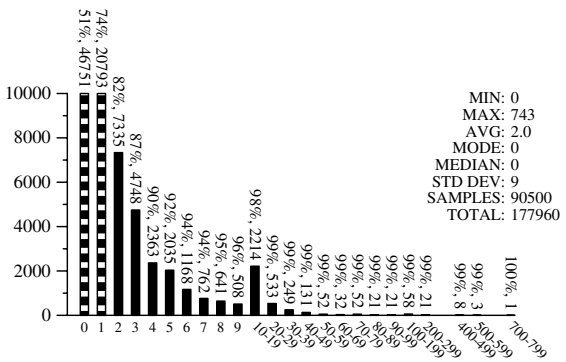
(b)

Figure 17. Method declarations in classes: (a) number of non-static methods per class; (b) number of final methods per class.

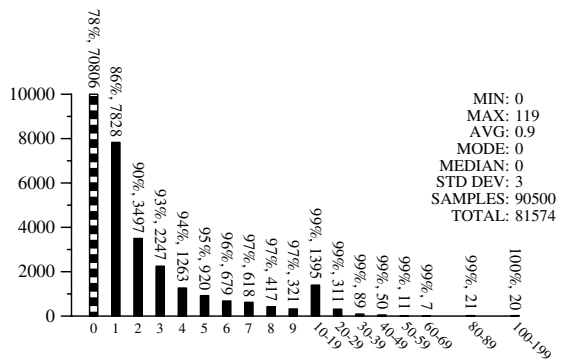
Table XI gives a breakdown of slot types. Note that Java's `short`, `byte`, `char`, and `boolean` types are compiled into integers in the bytecode, and thus will not show up as distinct types. Also, a slot may contain more than one type within a method, although at any one particular location it must always have the same type. Table XI shows that `int`s and `String`s make up the majority of slot types. Only 3.8% of slots contain two types, and only 0.6% contain three types. This indicates that the design of the JVM could have been simplified by requiring each slot to contain exactly one type throughout the body of a method, without much adverse effect.

Slots are not explicitly typed in the bytecode. Instead, slot types have to be computed using a static analysis known as *stack simulation*. This involves simulating the behavior of each instruction on the stack and the local variable slots, while following all possible paths of control flow within the method. A similar algorithm is used in the Java bytecode verifier.

DOI: 10.1002/spe



(a)



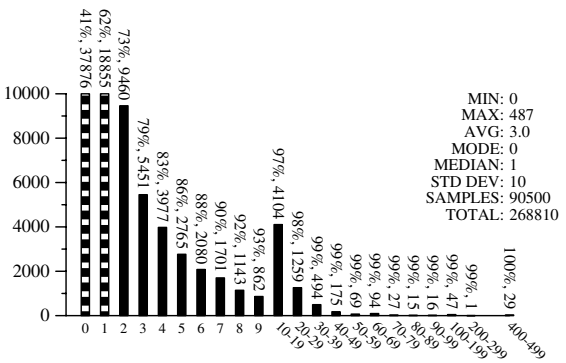
(b)

Figure 18. Protection of class members: (a) number of package private members per class; (b) number of private members per class.

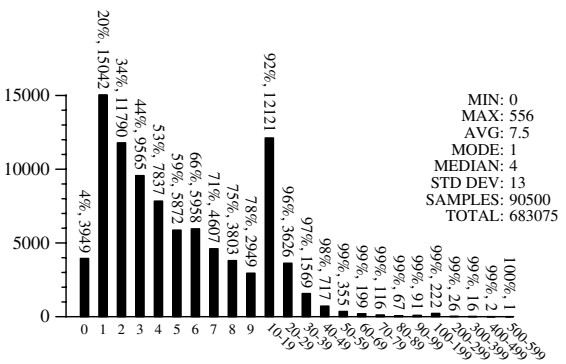
Figure 24(b) shows the maximum stack depth required by a method. This is stored as an attribute in the class file, and could thus be larger then the *actual* stack size needed at runtime.

The number of slots used by a method in Figure 24(a) includes those slots reserved for method parameters. Figure 25 breaks out the number of formal parameters per method. This is the number of parameters, not the number of slots those parameters would consume (i.e. longs and doubles count as one). As expected, the average is low (1.0), with 90% of all methods having two or fewer formals. Table XII shows the most common method signatures. The signatures are presented with the method's parameter type list first, followed by the method's return type. So, for example, a method that takes two integer parameters and returns a String would have a signature of '`(int,int)java.lang.String`'. We have also abstracted away any reference types that do not appear in the standard Java libraries (i.e. user-defined classes). If a user-defined class is a parameter or the return type of a method, we replace it with '`user_class`'. One reason that '`()void`' is so

DOI: 10.1002/spe



(a)



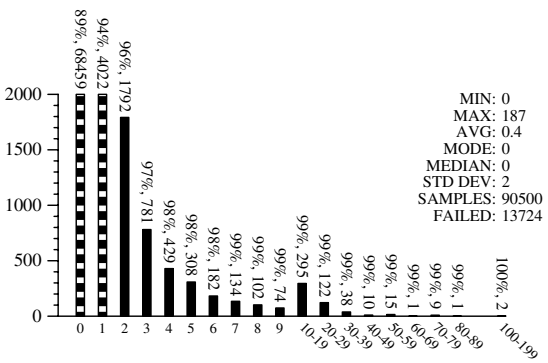
(b)

Protection	%
Package private	14.7
Private	6.7
Protected	22.2
Public	56.4

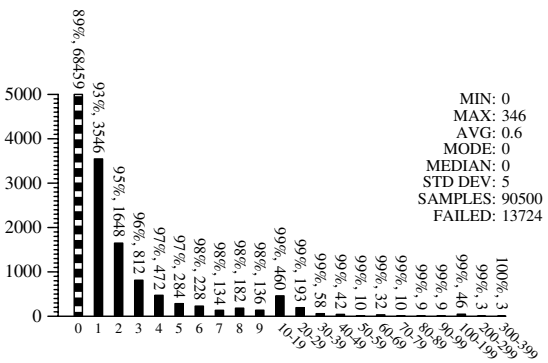
(c)

Figure 19. Protection of class members: (a) number of protected members per class; (b) number of public methods per class; (c) average of class members with particular protection.

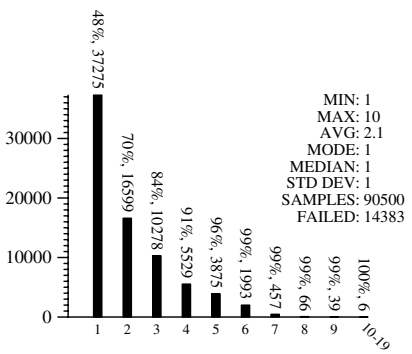
DOI: 10.1002/spe



(a)



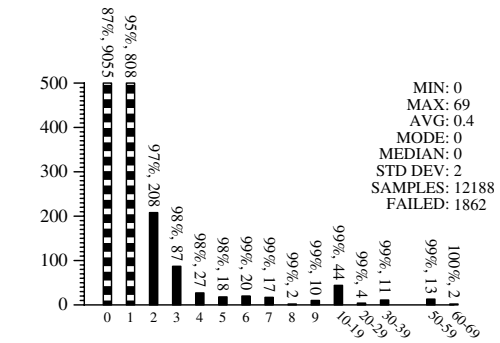
(b)



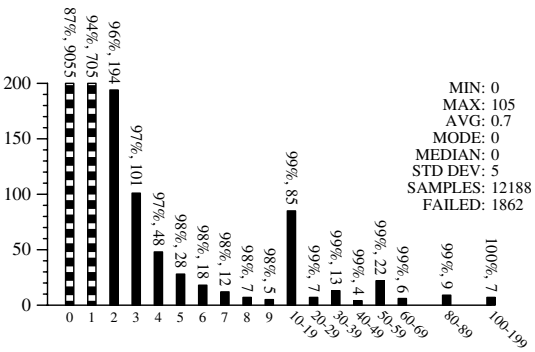
(c)

Figure 20. (a) Number of immediate subclasses per class; (b) total number of subclasses per class; (c) inheritance depth of a class.

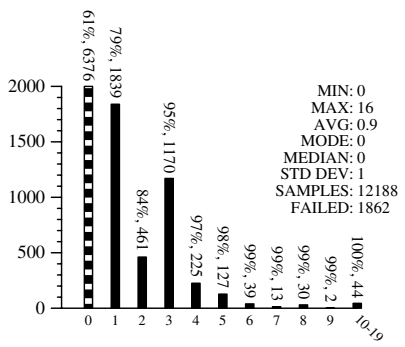
DOI: 10.1002/spe



(a)



(b)



(c)

Figure 21. (a) Number of immediate subinterfaces per interface; (b) total number of subinterfaces per interface; (c) interface extends depth of a class.

DOI: 10.1002/spe

Table IX. Most common standard interfaces to be extended by application interfaces.

Interface	Count	%
user_interface	3359	57.7
org.w3c.dom.html.HTMLElement	676	11.6
java.util.EventListener	362	6.2
java.io.Serializable	251	4.3
org.w3c.dom.Node	225	3.9
java.lang.Cloneable	118	2.0
org.omg.CORBA.Object	96	1.6
java.security.PrivateKey	43	0.7
org.w3c.dom.CharacterData	42	0.7
org.w3c.dom.events.EventTarget	39	0.7
java.security.PublicKey	39	0.7
org.omg.CORBA.portable.IDLEntity	38	0.7
org.omg.CORBA.IDLType	36	0.6
org.w3c.dom.Element	29	0.5
org.w3c.dom.Document	28	0.5
java.rmi.Remote	24	0.4
org.w3c.dom.css.CSSRule	23	0.4
java.security.Key	23	0.4
org.w3c.dom.events.Event	22	0.4
org.w3c.dom.DOMImplementation	22	0.4
org.w3c.dom.Text	21	0.4
org.xml.sax.XMLReader	20	0.3
org.omg.CORBA.IRObject	18	0.3
org.xml.sax.ContentHandler	16	0.3
java.lang.Comparable	16	0.3
javax.crypto.interfaces.DHKey	14	0.2
java.util.Map	12	0.2
java.sql.ResultSet	11	0.2
java.util.List	10	0.2
java.sql.Connection	9	0.2
java.lang.Runnable	9	0.2
org.w3c.dom.css.CSSValue	8	0.1
org.xml.sax.Locator	7	0.1
org.xml.sax.DTDHandler	7	0.1
java.util.Collection	7	0.1
java.sql.ResultSetMetaData	7	0.1
org.xml.sax.ext.LexicalHandler	6	0.1
org.omg.CORBA.DynAny	6	0.1
org.xml.sax.DocumentHandler	5	0.1
org.omg.CORBA.Policy	5	0.1
javax.xml.transform.SourceLocator	5	0.1
java.sql.PreparedStatement	5	0.1
org.w3c.dom.events.UIEvent	4	0.1
org.w3c.dom.events.DocumentEvent	4	0.1

DOI: 10.1002/spe

Table X. Most common standard interfaces to be implemented by application classes.

Interface	Count	%
user_interface	21 955	55.9
java.io.Serializable	3534	9.0
java.awt.event.ActionListener	2880	7.3
java.lang.Runnable	1447	3.7
java.lang.Cloneable	1009	2.6
org.omg.CORBA.portable.Streamable	793	2.0
java.awt.event.ItemListener	302	0.8
java.lang.Comparable	266	0.7
java.util.Iterator	262	0.7
java.util.Enumeration	216	0.6
java.util.Comparator	215	0.5
javax.swing.event.ChangeListener	211	0.5
java.awt.event.MouseListener	187	0.5
org.xml.sax.EntityResolver	173	0.4
java.security.PrivilegedAction	145	0.4
org.xml.sax.ErrorHandler	130	0.3
java.security.spec.AlgorithmParameterSpec	130	0.3
java.beans.PropertyChangeListener	114	0.3
java.awt.event.MouseMotionListener	113	0.3
org.xml.sax.ext.LexicalHandler	109	0.3
java.awt.event.KeyListener	109	0.3
org.xml.sax.ContentHandler	100	0.3
javax.swing.event.ListSelectionListener	99	0.3
java.io.Externalizable	99	0.3
java.security.spec.KeySpec	87	0.2
org.xml.sax.DocumentHandler	83	0.2
org.xml.sax.DTDHandler	82	0.2
java.awt.event.AdjustmentListener	81	0.2
javax.sql.DataSource	80	0.2
java.awt.event.WindowListener	80	0.2
java.awt.image.ImageObserver	76	0.2
java.awt.image.renderable.RenderedImageFactory	74	0.2
javax.naming.spi.ObjectFactory	72	0.2
java.sql.Connection	71	0.2
java.awt.event.FocusListener	71	0.2
org.w3c.dom.NodeList	70	0.2
org.xml.sax.AttributeList	59	0.2
javax.naming.Referenceable	58	0.1
java.io.FilenameFilter	55	0.1
org.xml.sax.Locator	52	0.1
java.util.Map$Entry	52	0.1
java.lang.reflect.InvocationHandler	52	0.1
javax.swing.event.DocumentListener	51	0.1
java.awt.event.ComponentListener	50	0.1
org.xml.sax.Attributes	48	0.1

DOI: 10.1002/spe

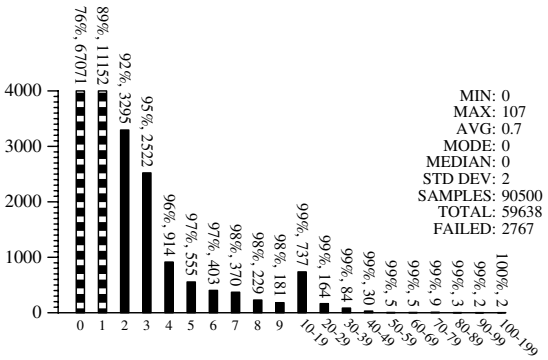


Figure 22. Number of method overrides per class.

common is that this is the signature of default constructors, especially that for `java.lang.Object`, which must be called in the constructors of all classes that directly extend it.

6.3. CFGs

A method body can be converted into a CFG, where the nodes (the *basic blocks*) are straight-line pieces of code. Control always enters the top of the basic block and exits at the bottom, either through an explicit branch or by *falling through* to another block. There is an edge from basic block A to basic block B if control can flow from A to B.

Building CFGs for Java bytecode is not straightforward. A major complication is how to deal with exception handling. Several instructions in the JVM can throw exceptions implicitly. This includes the division instructions (which may throw a divide-by-zero exception), and the `getfield`, `putfield`, and `invokevirtual` instructions (which may throw `null`-reference exceptions). These changes in control flow can be represented by adding *exception edges* to the CFG, which connect a basic block ending in an exception-throwing instruction to the CFG's sink node. If every such instruction (which are very common in real code) is allowed to terminate a basic block, blocks become very small. Since some analyses can safely ignore implicit exceptions, SandMark supports building the CFGs both with and without implicit exception edges. The `jsr` and `ret` instructions used for Java's `finally` clause also cause problems. In general, a data flow analysis is necessary in order to correctly build CFGs in the presence of complex `jsr`/`ret` combinations. SandMark currently does not support this and, as a consequence, will sometimes introduce spurious edges out of blocks ending in `ret` instructions. Since there are few such CFGs in our sample set this problem is unlikely to significantly affect our data.

Figure 26 shows the number of basic blocks per method body (we make the distinction *method body* to rule out methods with no instructions, such as native or abstract methods). We can see that 97% of all method bodies have fewer than 100 basic blocks. We can corroborate this information with Figures 27(a) and 23(b), to see that the average basic block size is 2.0, and that 97% of all method bodies have fewer than 200 instructions.

DOI: 10.1002/spe

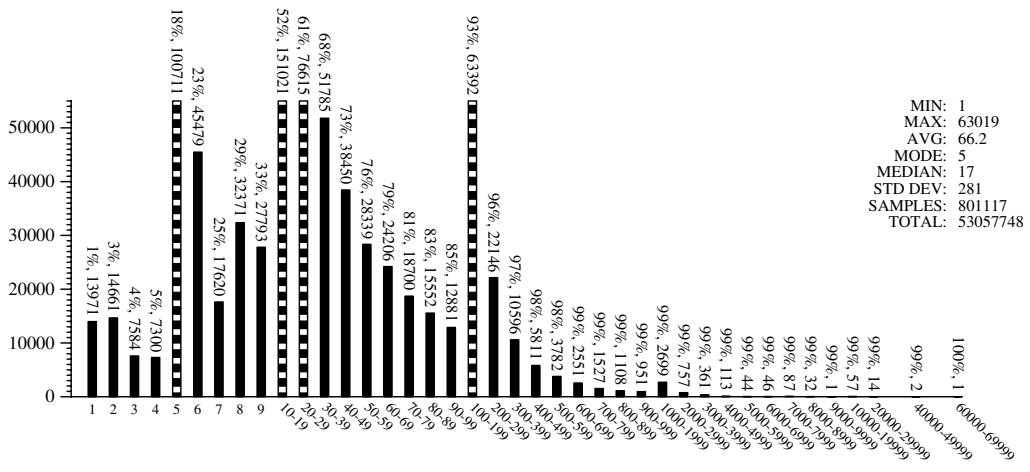


Figure 23. Method sizes: (a) size in bytes; (b) number of instructions per method body.

As can be seen from Figure 27(a), the average number of instructions in a basic block is very small, only 2.0, and 98% of all blocks have fewer than six instructions. Figure 28(a) shows the average out-degree of a basic block node is, predictably, low, only 1.2. Out-degrees higher than two can only be achieved either when an instruction is inside an exception handler's `try` block, or with the JVM's `tableswitch` and `lookupswitch` instructions. In the first case, edges are added from each basic block inside a `try` block to the first basic block of the handler code. Therefore, if a basic block is inside multiple nested `try` blocks its out-degree may be high. In the second case,

DOI: 10.1002/spe

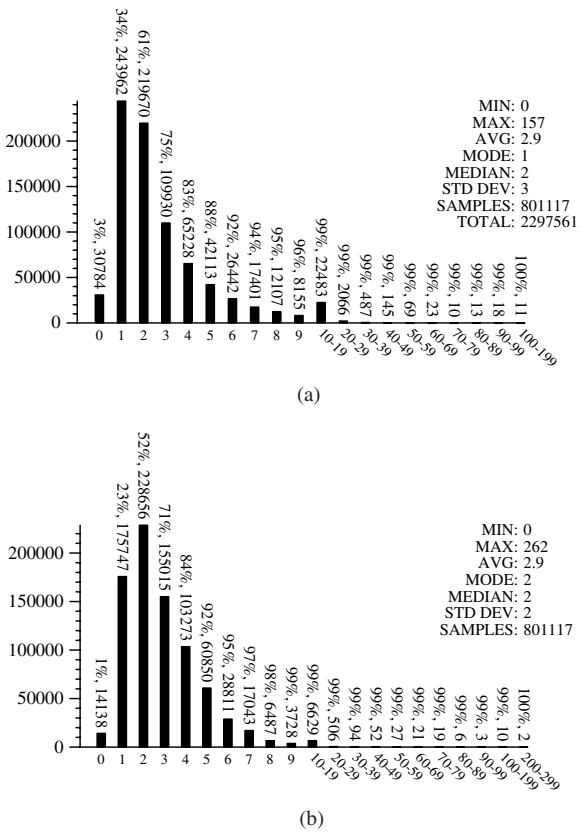


Figure 24. Local variables: (a) number of max locals per method body; (b) number of max stack weights per method body.

the `tableswitch` and `lookupswitch` instructions are Java's implementation of `switch`-statements, which may have many possible branch targets. Higher in-degrees can occur when a `try` catch block has many instructions inside it that could potentially trigger the exception. Each of these instructions will end its block, and have an edge going from it to the handler block. Thus, the in-degree of the handler block will become large.

Figure 27(b) shows the number of instructions per basic block when implicit exception edges have not been generated. As can be seen, this increases the average number of instructions per block to 7.7.

A node x in a directed graph G with a single exit node dominates node y in G if every path from the entry node to y must pass through x. The dominator set of a node y is the set of all nodes which dominate y. Dominator information is used in code optimizations such as loop identification and code motion. Figure 29 shows the number of dominator blocks per basic block.

DOI: 10.1002/spe

Table XI. Most common slot types.

Register type	Count	%
`int`	614 910	16.2
`java.lang.String`	365 915	9.6
2 types	144 145	3.8
`java.lang.Object`	76 764	2.0
`byte[]`	50 658	1.3
`long`	49 903	1.3
`java.lang.Throwable`	38 046	1.0
`double`	25 541	0.6
3 types	23 426	0.6
`java.lang.StringBuffer`	21 716	0.6
`java.lang.String[]`	20 600	0.5
`java.util.Iterator`	16 036	0.4
`float`	15 595	0.4
`java.lang.Class`	15 129	0.4
`java.util.Vector`	14 795	0.4
`int[]`	14 604	0.4
`java.lang.Exception`	14 149	0.4
`java.io.File`	13 334	0.4
`java.io.InputStream`	11 686	0.3
`java.util.List`	11 615	0.3
`java.lang.ClassNotFoundException`	11 331	0.3
`java.util.Enumeration`	10 732	0.3
`char[]`	9534	0.3
`java.lang.Object[]`	9417	0.2

6.4. Subroutines and exception handlers

Java subroutines are implemented by the instructions `jsr` and `ret`. They are chiefly used to implement the `finally` clause of an exception handler. This clause can be reached from multiple locations. For example, a `return` instruction within the body of a `try` block will first jump to the `finally` clause before returning from the method. Similarly, before returning from within an exception handler, the `finally` block must be executed. To avoid code duplication (inlining the `finally` block at every location from which it could be called) the designers of the JVM added the `jsr` and `ret` instructions to jump to and return from a block of code. This has caused much complication in the design of the JVM verifier. See, for example, Stata *et al*. [7]. Figure 30 shows that 98% of methods have no more than two exception handlers, and 98% of all methods have no subroutines. Figure 30(c) shows that the average size of a subroutine is 7.5 instructions. The length of a subroutine was computed as the number of instructions between a `jsr`'s target and its corresponding `ret`. Together, our data indicate that `jsr` and `ret` could have been left out of the JVMs instruction set without much code increase from `finally` clauses being implemented by code duplication.

DOI: 10.1002/spe

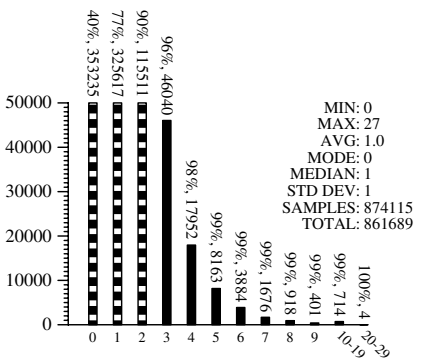


Figure 25. Number of formal parameters per method.

6.5. Interference graphs

An interference graph models the variables and live range interferences of a method. The live ranges of a local variable are the locations in a method between where the variable is first assigned and where it is last used. Since method parameters and the '`this`' reference are in local variable slots, they are considered to have their first assignment before the first instruction of the method. The graph has one vertex per local variable and an edge between two vertices when the corresponding variables' live ranges overlap (or *interfere*). As an example consider the sample code in Figure 31(a) and the corresponding interference graph in Figure 31(b). Since the code has 5 variables, the graph has 5 nodes. The graph has an edge $v_1 \rightarrow v_2$ since variables v_1 and v_2 are live at the same time. An interference graph is often used during the code generation pass of a compiler to perform register allocation. Two variables with intersecting live ranges cannot be assigned to the same register. Figure 32 shows that 95% of the methods have nine or fewer nodes in their interference graphs. This means that methods typically will need very few local variable slots. This analysis agrees with the data in Figure 24(a), which show that methods declare their maximum number of slots to be small.

After examining these data, it appears that the designers of the Java instruction set were wise to make the typical instruction use only 1 byte to refer to a local variable index. The `wide` prefix allows such an instruction to use a 2-byte index, but as we can see this will almost never be necessary. Thus, had the designers simply made all instructions use 2-byte indices, there would have been much wasted space in the bytecode.

7. INSTRUCTION-LEVEL STATISTICS

In this section we present information regarding the frequency of individual instructions and patterns of instructions. We also show the most common subexpressions and constant values found in the bytecode.

DOI: 10.1002/spe

Table XII. Most common method signatures.

Method signature	Count	%
()void	120 997	13.8
(*user_class*)void	57 762	6.6
()java.lang.String	53 047	6.1
()*user_class*	44 098	5.0
(java.lang.String)void	43 810	5.0
()boolean	39 772	4.5
()int	35 064	4.0
(int)void	18 959	2.2
(boolean)void	11 461	1.3
(*user_class*)*user_class*	10 332	1.2
(*user_class*,*user_class*)void	9652	1.1
(java.lang.String)*user_class*	7781	0.9
(java.lang.String)java.lang.String	7777	0.9
(*user_class*)boolean	6880	0.8
(*user_class*)java.lang.Object	6812	0.8
()java.lang.Object	6461	0.7
(java.lang.String)java.lang.Class	6258	0.7
(java.lang.String,java.lang.String)void	5561	0.6
(java.lang.Object)boolean	5373	0.6
(int)int	4776	0.5
(java.lang.Object)void	4697	0.5
(java.awt.event.ActionEvent)void	4479	0.5
(int)*user_class*	4270	0.5
(int)boolean	4116	0.5
(java.lang.String[])void	4044	0.5
(java.lang.String)boolean	3933	0.4
(int,int)void	3726	0.4
(int)java.lang.String	3473	0.4
()byte[]	3380	0.4
()*user_class*[]	3322	0.4
(*user_class*,int)void	3251	0.4
()java.util.List	2970	0.3
(*user_class*,java.lang.String)void	2821	0.3
(byte[])void	2697	0.3
()java.lang.String[]	2292	0.3
(*user_class*,*user_class*)*user_class*	2289	0.3
(int,int)int	2023	0.2
(java.awt.event.MouseEvent)void	2008	0.2
(*user_class*)int	1998	0.2
(java.lang.String)int	1993	0.2
(*user_class*)java.lang.String	1951	0.2
()org.omg.CORBA.TypeCode	1941	0.2
(java.lang.String,*user_class*)void	1900	0.2
(java.lang.Object)*user_class*	1873	0.2

DOI: 10.1002/spe

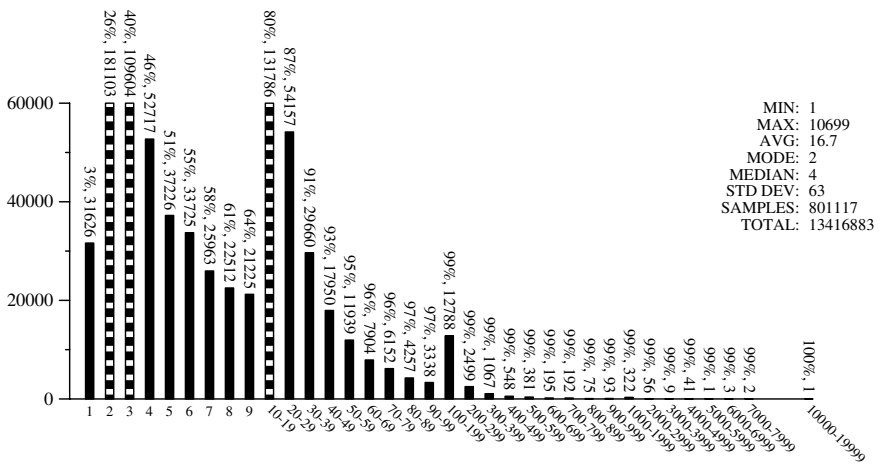


Figure 26. Number of basic blocks (in CFGs with implicit exception edges) per method body.

7.1. Instruction counts

There are 200 usable JVM instruction opcodes. Table XIII shows the frequency of each of those bytecode instructions. The most frequently occurring instruction is `aload_0` which is responsible for pushing the local variable 0, the `this` reference of non-static methods. Even though this is the most frequently occurring instruction it only has a frequency of 10%. The `invokevirtual` instruction which calls a non-static method is also common, as is `getfield`, `dup`, and `invokespecial`, the last two being used to implement Java's `new` operator. These five instructions account for 33.8% of all instructions. Our data indicate that the majority of the remaining instructions each occur with a frequency of at most 1%, and that the `jsr_w` and `goto_w` instructions (used for long branches) do not occur at all.

7.2. Instruction patterns

A k-gram is a contiguous substring of length k which can be comprised of letters, words, or, in our case, opcodes. The k-gram is based on static analysis of the executable program. To compute the unique set of k-grams for a method, we slide a window of length k over the static instruction sequence as it is laid out in the class file.

We computed data for k-grams where $k = 2, 3, 4$, which is shown in Tables XIV, XV, and XVI, respectively. These tables show that as the value of k increases the percentages of the most frequently occurring sequences decrease. For example, the most frequently occurring 2-gram, `aload_0, getfield`, has a frequency of only 4.7%. For 3- and 4-grams, the most frequently occurring sequence is less than 1%. This indicates that these sequences become quite unique for each individual application.

DOI: 10.1002/spe

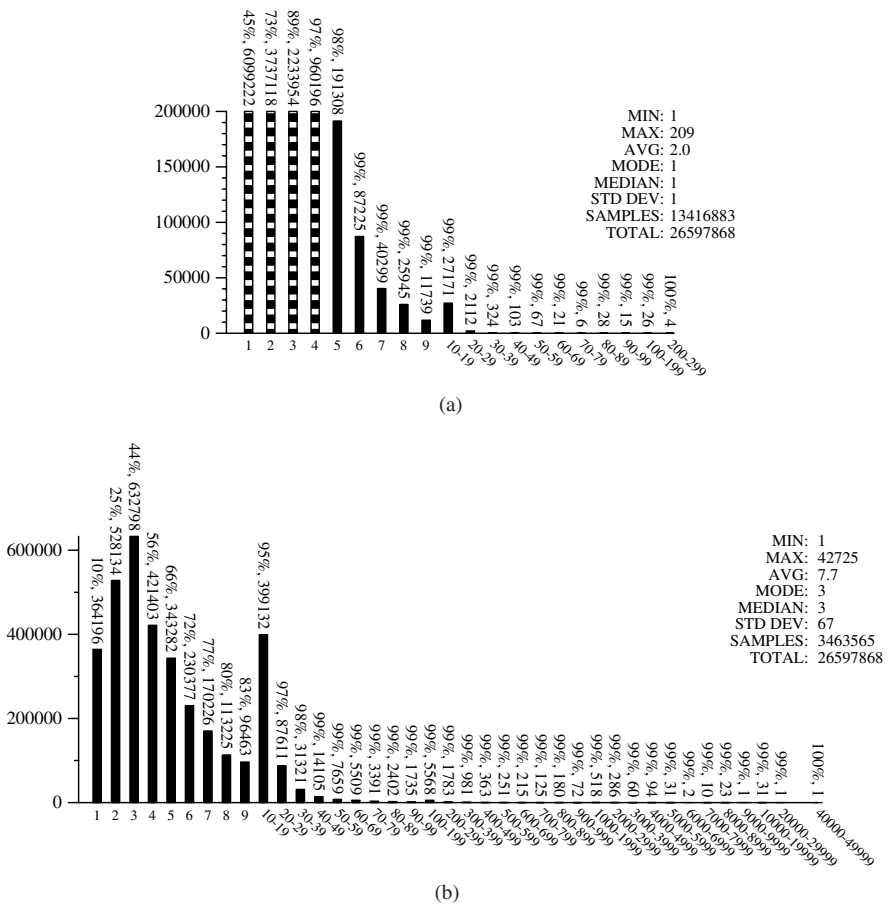


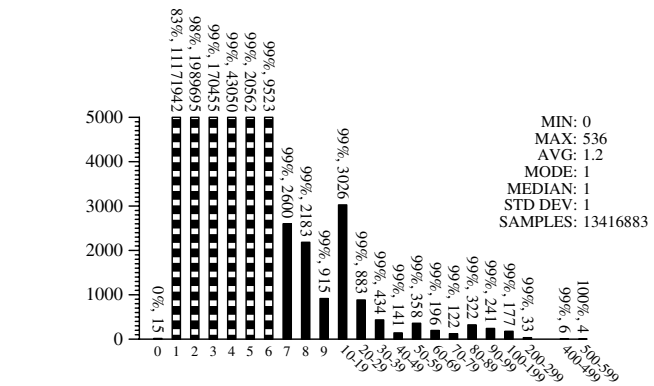
Figure 27. Number of instructions per basic block in CFGs (a) with and (b) without implicit exception edges.

7.3. Expressions

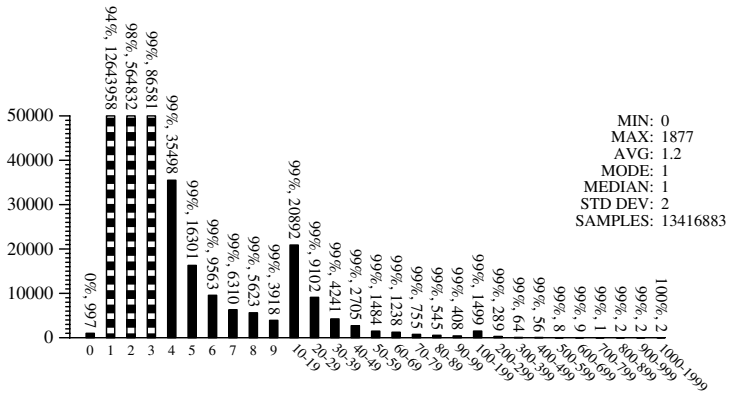
Figure 33 shows the size (number of nodes in the tree) and height (length of longest path from root to leaf) of expression trees in our samples. As reported already by Knuth [1], expressions tend to be small. We found that 61% of all expressions only have one node.

Expressions are constructed by performing a stack simulation over each method. For each instruction that will produce a result on the stack the simulator determines which instructions may have put its operands on the stack. This information is used to build up a dependency graph with instructions and operands as nodes, and an edge from node a to node b if b is used by a. If the program contains certain

DOI: 10.1002/spe



(a)



(b)

Figure 28. CFGs: (a) out-degree of basic block nodes; (b) in-degree of basic block nodes.

types of loops these graphs might have cycles, in which case they are discarded. The following code segment is an example of such a loop:

```
0: ICONST_2
1: ICONST_3
2: IADD
3: DUP
4: IFEQ <1>
```

In this example, the `IADD` instruction may become its own child. If the branch at offset 4 is taken, then the result from the first iteration of the `IADD` will be used as the first operand in the second iteration of the `IADD`.

DOI: 10.1002/spe

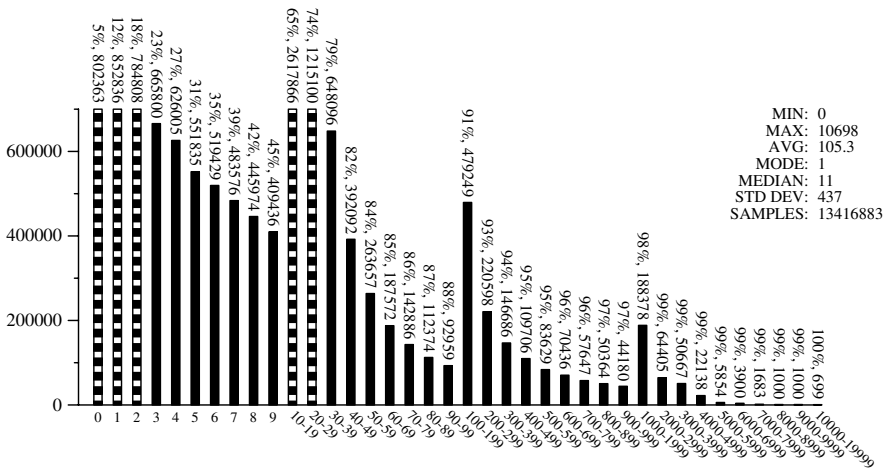


Figure 29. Number of dominator blocks per basic block (in CFGs with implicit exception edges).

In Table XVII we show the most common subexpressions found in our sample method bodies. Table XVIII explains the abbreviations used. `L`, for example, represents a local variable, `d` a double constant, (α+α) addition, etc. Since we are counting subexpressions, the same piece of an expression will be counted more than once. For example, if `x` and `y` are local variables, then the expression `x+y*2` will generate subexpressions `L`, `L`, `i`, `(L*i)`, and `(L+(L*i))`, each of which will increase the count of its respective expression class.

To compute subexpressions we convert each expression tree into a string representation, classify each subexpression into equivalence classes according to Table XVIII, and count each subexpression individually.

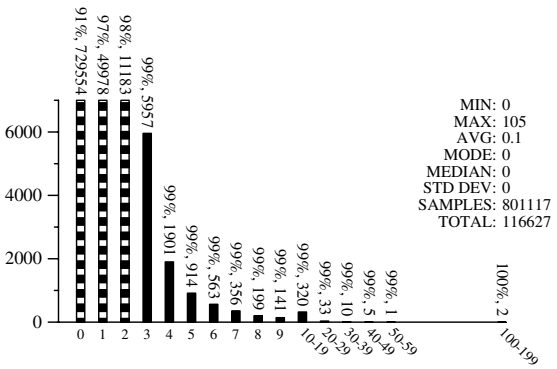
What we see from Table XVII is that, unsurprisingly, local variable references, method calls, integer constants, and field references make up the bulk of expressions. Somewhat more surprising is that the expression `((Class)M())` is very frequent. Most likely this is the result of references to 'generic' methods (particularly Java library functions such as `java.util.Vector.get()`) returning `java.lang.Objects` which then have to be cast into a more specific type.

7.4. Constant values

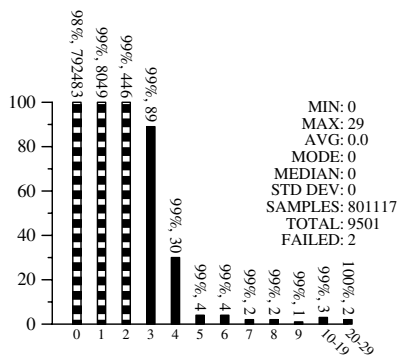
Tables XIX, XX, and XXI show the most common literal constants found in the bytecode. Constants can occur in three different ways: as references to entries in the constant pool (instructions `ldc`, `ldc_w`, and `ldc2_w`), as arguments to bytecode instructions (`bipush` n, `sipush` n, and `iinc` n,c), or embedded in special instructions (`iconst_`n, etc.).

Figure 34 shows the distribution of integer constant values. It is interesting to note that 63% of all literal integers are 0, powers of two, or powers of two plus/minus one. This has implications for,

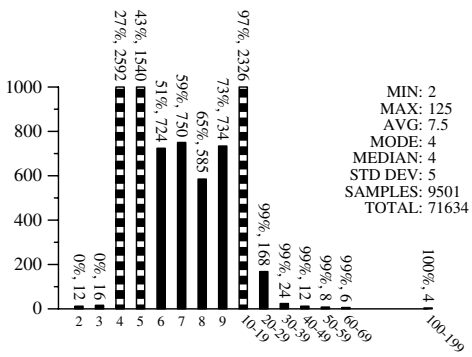
DOI: 10.1002/spe



(a)



(b)



(c)

Figure 30. (a) Number of exception handlers per method body; (b) number of subroutines per method body; (c) number of instructions per subroutine.

DOI: 10.1002/spe

Figure 31. (a) Sample code and (b) corresponding interference graph.

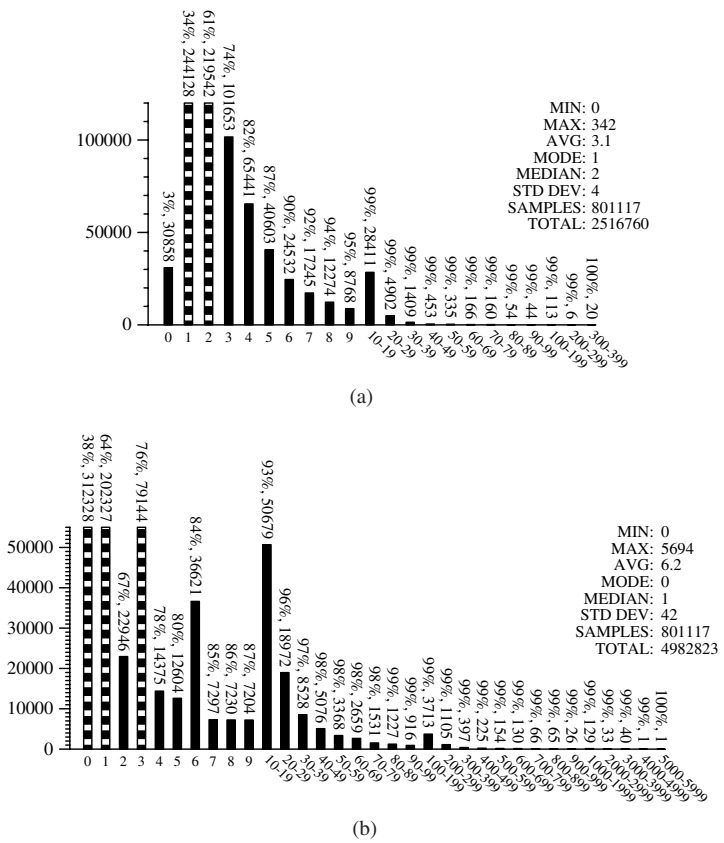


Figure 32. (a) Number of interference graph nodes per method body; (b) number of interference graph edges per method body.

DOI: 10.1002/spe

Table XIII. Instruction frequencies.

Opcode	Count	%	Opcode	Count	%
aload_0	2 672 134	10.0	aaload	108 016	0.4
invokevirtual	2 360 924	8.9	anewarray	106 780	0.4
dup	1 521 855	5.7	putstatic	105 900	0.4
getfield	1 447 792	5.4	astore_1	99 436	0.4
invokespecial	1 003 439	3.8	isub	93 852	0.4
ldc	936 890	3.5	if_icmplt	89 901	0.3
aload_1	909 356	3.4	if_icmpne	89 452	0.3
aload	876 138	3.3	iconst_3	85 851	0.3
bipush	865 346	3.3	arraylength	81 083	0.3
new	665 727	2.5	iaload	70 687	0.3
iconst_0	634 481	2.4	iconst_m1	67 600	0.3
iload	601 808	2.3	ldc2_w	66 717	0.3
putfield	552 241	2.1	iconst_4	64 544	0.2
goto	507 322	1.9	istore_3	58 529	0.2
iconst_1	495 114	1.9	istore_2	57 750	0.2
aload_2	494 004	1.9	iand	55 782	0.2
invokestatic	457 014	1.7	instanceof	50 049	0.2
getstatic	438 851	1.6	if_acmpne	48 379	0.2
return	433 081	1.6	if_icmpeq	47 866	0.2
astore	395 436	1.5	newarray	44 390	0.2
sipush	383 115	1.4	iconst_5	39 857	0.1
areturn	351 978	1.3	baload	39 482	0.1
aastore	332 112	1.2	castore	38 065	0.1
aload_3	314 398	1.2	istore_1	37 068	0.1
invokeinterface	300 563	1.1	if_icmpge	35 921	0.1
ifeq	286 898	1.1	sastore	30 690	0.1
iastore	285 979	1.1	ixor	29 811	0.1
ldc_w	281 190	1.1	if_icmple	28 212	0.1
pop	270 894	1.0	imul	27 174	0.1
istore	264 341	1.0	iload_0	26 837	0.1
ireturn	259 627	1.0	dload	26 640	0.1
iload_2	200 600	0.8	lastore	23 738	0.1
iload_1	197 241	0.7	ifle	23 025	0.1
checkcast	193 243	0.7	monitorexit	22 023	0.1
aconst_null	178 499	0.7	jsr	20 074	0.1
iload_3	172 820	0.6	lconst_0	19 617	0.1
bastore	171 902	0.6	nop	18 136	0.1
iadd	171 637	0.6	lload	17 515	0.1
ifne	167 878	0.6	ifge	17 494	0.1
iconst_2	163 348	0.6	i2b	17 432	0.1
athrow	151 515	0.6	ishl	17 233	0.1
astore_2	144 741	0.5	fload	16 966	0.1
iinc	132 890	0.5	ior	15 500	0.1
astore_3	121 477	0.5	ishr	15 363	0.1
ifnull	121 318	0.5	lcmp	15 033	0.1
ifnonnull	110 290	0.4	dstore	14 261	0.1

DOI: 10.1002/spe

Table XIII. *Continued*.

Opcode	Count	%	Opcode	Count	%	Opcode	Count	%
dup_x1	14 202	0.1	fconst_0	4316	0.0	fneg	496	0.0
dmul	14 077	0.1	f2d	4086	0.0	pop2	378	0.0
idiv	13 833	0.1	laload	3781	0.0	fstore_1	374	0.0
if_icmpgt	12 477	0.0	dload_3	3538	0.0	d2l	273	0.0
iflt	11 944	0.0	lconst_1	3515	0.0	dup2_x1	263	0.0
caload	11 871	0.0	dload_2	3286	0.0	lneg	208	0.0
if_acmpeq	11 595	0.0	fload_1	3261	0.0	l2f	187	0.0
dastore	11 325	0.0	lsub	3212	0.0	dstore_0	168	0.0
astore_0	10 400	0.0	fload_2	3130	0.0	dup2_x2	164	0.0
tableswitch	10 197	0.0	dcmpg	3040	0.0	lstore_0	159	0.0
land	10 047	0.0	dup_x2	2984	0.0	drem	55	0.0
monitorenter	9961	0.0	fdiv	2930	0.0	f2l	42	0.0
daload	9782	0.0	saload	2867	0.0	fstore_0	20	0.0
fastore	9708	0.0	ineg	2577	0.0	frem	12	0.0
ret	9670	0.0	multianewarray	2500	0.0	jsr_w	0	0.0
dconst_0	9295	0.0	d2f	2402	0.0	goto_w	0	0.0
i2l	8832	0.0	freturn	2360	0.0			
fstore	8765	0.0	fload_3	2357	0.0			
lookupswitch	8738	0.0	lshl	2195	0.0			
lload_1	8723	0.0	fconst_1	2122	0.0			
faload	8634	0.0	dload_0	2093	0.0			
iushr	8468	0.0	lload_0	1982	0.0			
dadd	8407	0.0	fcmpl	1917	0.0			
i2d	8403	0.0	istore_0	1787	0.0			
ifgt	7976	0.0	lstore_3	1727	0.0			
fmul	7674	0.0	lmul	1664	0.0			
lstore	7512	0.0	lor	1520	0.0			
dup2	7332	0.0	lstore_2	1475	0.0			
lload_2	7125	0.0	f2i	1391	0.0			
ddiv	6644	0.0	lshr	1386	0.0			
lload_3	6514	0.0	lstore_1	1352	0.0			
dsub	5882	0.0	fcmpg	1243	0.0			
i2c	5839	0.0	dneg	1230	0.0			
l2i	5680	0.0	dstore_3	1215	0.0			
dload_1	5548	0.0	ldiv	1194	0.0			
fadd	5515	0.0	lxor	1146	0.0			
irem	5508	0.0	dstore_2	1085	0.0			
dreturn	5159	0.0	lushr	993	0.0			
dconst_1	5146	0.0	dstore_1	908	0.0			
dcmpl	5065	0.0	l2d	878	0.0			
i2f	5003	0.0	swap	849	0.0			
lreturn	4935	0.0	fconst_2	839	0.0			
ladd	4621	0.0	fstore_3	695	0.0			
fsub	4463	0.0	fstore_2	650	0.0			
i2s	4338	0.0	lrem	539	0.0			
d2i	4331	0.0	fload_0	510	0.0			

DOI: 10.1002/spe

Table XIV. Most common 2-grams.

Opcode	Count	%
aload_0,getfield	1 219 837	4.7
new,dup	664 718	2.6
ldc,invokevirtual	353 412	1.4
invokevirtual,invokevirtual	332 487	1.3
dup,bipush	330 887	1.3
putfield,aload_0	311 038	1.2
iastore,dup	250 744	1.0
invokevirtual,aload_0	235 924	0.9
dup,sipush	226 520	0.9
aload_1,invokevirtual	223 958	0.9
aload,invokevirtual	222 692	0.9
getfield,invokevirtual	219 107	0.8
aload_0,aload_1	214 369	0.8
aastore,dup	208 247	0.8
dup,invokespecial	202 840	0.8
aload_0,invokevirtual	200 872	0.8
invokevirtual,pop	193 105	0.7
aload_0,invokespecial	159 742	0.6
astore,aload	146 309	0.6
bastore,dup	141 779	0.5
ldc,aastore	133 994	0.5
getfield,aload_0	129 300	0.5
invokespecial,aload_0	122 168	0.5
ldc,invokespecial	120 935	0.5
dup,ldc	116 043	0.4
invokespecial,athrow	115 994	0.4
aload_2,invokevirtual	115 394	0.4
goto,aload_0	115 340	0.4
putfield,return	113 044	0.4
dup,iconst_0	109 473	0.4
invokevirtual,ldc	109 060	0.4
invokevirtual,return	106 765	0.4
invokevirtual,ifeq	103 093	0.4
bipush,bastore	102 969	0.4
invokevirtual,astore	100 355	0.4
ifeq,aload_0	99 667	0.4
bipush,bipush	98 715	0.4
ldc_w,iastore	98 376	0.4
iconst_0,ireturn	98 199	0.4
invokevirtual,aload	93 325	0.4
aload_0,new	90 040	0.3
anewarray,dup	81 992	0.3
dup,aload_0	80 579	0.3
aload_0,aload_0	80 329	0.3
aload,aload	78 718	0.3

DOI: 10.1002/spe

Table XV. Most common 3-grams.

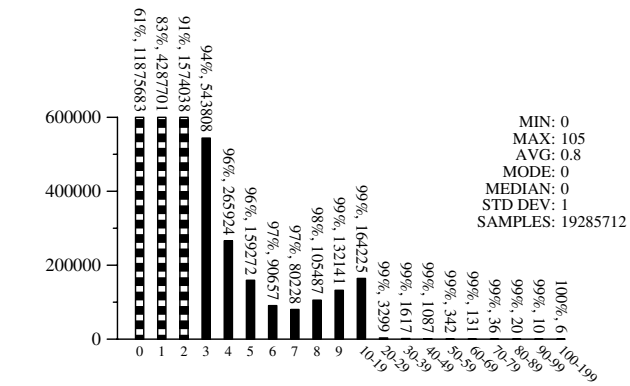
Opcode	Count	%
new,dup,invokespecial	202 836	0.8
aload_0,getfield,invokevirtual	194 765	0.8
iastore,dup,bipush	132 759	0.5
invokevirtual,aload_0,getfield	125 019	0.5
new,dup,ldc	115 036	0.5
aload_0,getfield,aload_0	111 950	0.4
getfield,aload_0,getfield	111 002	0.4
iastore,dup,sipush	102 667	0.4
bipush,bastore,dup	100 197	0.4
invokevirtual,ldc,invokevirtual	98 964	0.4
ldc_w,iastore,dup	97 826	0.4
dup,ldc,invokespecial	91 303	0.4
aload_0,new,dup	90 029	0.4
dup,bipush,bipush	83 402	0.3
ldc,aastore,dup	82 970	0.3
anewarray,dup,iconst_0	81 984	0.3
aastore,dup,bipush	80 740	0.3
new,dup,aload_0	80 524	0.3
invokevirtual,invokevirtual,invokevirtual	80 161	0.3
invokespecial,ldc,invokevirtual	69 626	0.3
aload_0,getfield,aload_1	68 922	0.3
bastore,dup,sipush	67 634	0.3
aload_0,invokespecial,aload_0	66 723	0.3
dup,sipush,bipush	64 736	0.3
aload_0,aload_1,putfield	60 661	0.2
bastore,dup,bipush	60 580	0.2
goto,aload_0,getfield	60 205	0.2
aload_0,aload_0,getfield	58 350	0.2
dup,invokespecial,ldc	57 764	0.2
dup,sipush,ldc_w	57 139	0.2
dup,bipush,ldc_w	56 309	0.2
aastore,dup,iconst_1	56 004	0.2
putfield,aload_0,getfield	55 324	0.2
aastore,aastore,dup	55 149	0.2
aload_0,getfield,areturn	55 073	0.2
ldc,invokevirtual,invokevirtual	53 465	0.2
new,dup,aload_1	52 185	0.2
ldc,invokevirtual,aload_0	51 342	0.2
invokespecial,putfield,aload_0	50 827	0.2
sipush,bipush,bastore	50 642	0.2
dup,bipush,ldc	50 439	0.2
dup,iconst_0,ldc	49 992	0.2
aload_0,getfield,getfield	49 974	0.2
iconst_0,ldc,aastore	49 056	0.2
sipush,ldc_w,iastore	48 252	0.2

DOI: 10.1002/spe

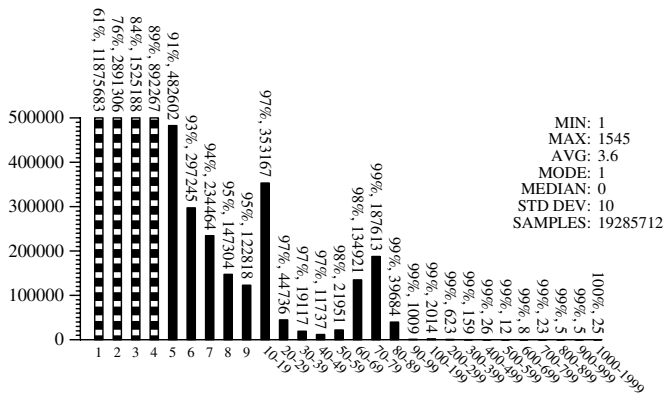
Table XVI. Most common 4-grams.

Opcode	Count	%
aload_0,getfield,aload_0,getfield	95 199	0.4
new,dup,ldc,invokespecial	91 302	0.4
new,dup,invokespecial,ldc	57 764	0.2
dup,invokespecial,ldc,invokevirtual	57 239	0.2
bipush,bastore,dup,sipush	50 663	0.2
dup,sipush,bipush,bastore	50 642	0.2
bastore,dup,sipush,bipush	50 642	0.2
sipush,bipush,bastore,dup	50 392	0.2
anewarray,dup,iconst_0,ldc	48 862	0.2
dup,iconst_0,ldc,aastore	48 697	0.2
iastore,dup,sipush,ldc_w	48 252	0.2
dup,sipush,ldc_w,iastore	48 252	0.2
ldc_w,iastore,dup,sipush	48 198	0.2
sipush,ldc_w,iastore,dup	47 875	0.2
iastore,dup,bipush,ldc_w	47 528	0.2
dup,bipush,ldc_w,iastore	47 528	0.2
ldc_w,iastore,dup,bipush	47 520	0.2
bipush,ldc_w,iastore,dup	47 384	0.2
dup,bipush,bipush,bastore	44 682	0.2
bastore,dup,bipush,bipush	44 680	0.2
bipush,bastore,dup,bipush	44 674	0.2
bipush,bipush,bastore,dup	44 209	0.2
aload_0,new,dup,invokespecial	43 141	0.2
new,dup,new,dup	42 678	0.2
invokevirtual,ldc,invokevirtual,invokevirtual	42 594	0.2
ldc,aastore,aastore,dup	41 430	0.2
aastore,aastore,dup,bipush	40 443	0.2
dup,ldc,invokespecial,athrow	40 441	0.2
new,dup,aload_0,getfield	36 325	0.1
new,dup,invokespecial,putfield	34 800	0.1
ldc,aastore,dup,iconst_1	34 705	0.1
iconst_0,ldc,aastore,dup	34 705	0.1
aastore,dup,iconst_1,ldc	34 585	0.1
putfield,aload_0,new,dup	34 499	0.1
dup,iconst_1,ldc,aastore	34 191	0.1
invokevirtual,aload_0,getfield,invokevirtual	34 185	0.1
aload_0,iconst_0,putfield,aload_0	33 108	0.1
aload_0,aload_1,putfield,return	32 147	0.1
aload_0,aconst_null,putfield,aload_0	31 719	0.1
aload_0,getfield,aload_1,invokevirtual	31 472	0.1
iconst_2,anewarray,dup,iconst_0	30 710	0.1
ldc,invokevirtual,aload_0,getfield	30 470	0.1
putfield,aload_0,iconst_0,putfield	27 739	0.1
iastore,dup,bipush,ldc	26 735	0.1
dup,bipush,ldc,iastore	26 735	0.1

DOI: 10.1002/spe



(a)



(b)

Figure 33. (a) Height and (b) size of expression trees.

for example, software watermarking algorithms such as that by Cousot and Cousot [11], which hides a watermark in unusual constants. Figure 34 tells us that in real programs most constants are small (93% are less than 1000) or very close to powers of two, and hence hiding a mark in unusual constants is likely to be unstealthy.

7.5. Method calls

Table XXII reports the most frequently called Java library methods. To collect these data, we looked at every `INVOKE` instruction to see what method it named. No attempt at method resolution was made.

Figure 35 measures the size of receiver sets of method calls. That is, for a virtual method invocation `o.M()` we count the number of methods `M()` that might potentially be called. This depends on the

DOI: 10.1002/spe

Table XVII. Most common subexpressions.

Expression	Count	%	Expression	Count	%
L	6 574 522	34.9	L.F.F.F	5087	0.0
M()	4 119 506	21.9	(L.F+L)	4607	0.0
i	2 967 193	15.7	(L.F&i)	4572	0.0
L.F	1 366 327	7.3	(M()+i)	4511	0.0
"	1 039 672	5.5	(L.F+L.F)	4326	0.0
N	665 727	3.5	(L&l)	4149	0.0
S	438 851	2.3	((L+M())+L.F[])	4128	0.0
A	153 670	0.8	(M()+L)	3997	0.0
L[]	121 966	0.6	(L.length-i)	3994	0.0
((*Class*)M())	115 363	0.6	((L>>i)&i)	3917	0.0
null	95 366	0.5	(i*L)	3792	0.0
L.F[]	83 259	0.4	((L&l)<>l)	3734	0.0
l	67 088	0.4	(L/i)	3689	0.0
L.F.F	54 078	0.3	(L.F>>i)	3654	0.0
L.length	51 938	0.3	(((L+M())+L.F[])+i)	3392	0.0
((*Class*)L)	49 937	0.3	L.F[].F	3367	0.0
(L+i)	41 182	0.2	S[][]	3351	0.0
(L instanceof *Class*)	39 081	0.2	(L.F instanceof *Class*)	3342	0.0
d	37 202	0.2	((L>>>i)&i)	3162	0.0
S[]	29 776	0.2	(L+L.F)	3051	0.0
(L+L)	24 534	0.1	((*Class*)L[])	2995	0.0
(L.F+i)	24 296	0.1	(L.F-L)	2977	0.0
L.F.length	23 898	0.1	(S[]&i)	2898	0.0
(L-i)	19 852	0.1	(L^L)	2818	0.0
f	17 746	0.1	(L.F[]&i)	2773	0.0
((*Class*)S)	14 614	0.1	((long)L)	2748	0.0
(L-L)	13 495	0.1	((byte)L)	2699	0.0
(L&i)	13 436	0.1	(L.F*L.F)	2690	0.0
(L.F-i)	10 759	0.1	S.length	2626	0.0
(L>>i)	9050	0.0	(l&L)	2618	0.0
(L[]&i)	8285	0.0	((l&L)<>l)	2612	0.0
((*Class*)L.F)	7715	0.0	((double)L.F)	2509	0.0
M().F	7518	0.0	((L[]&i)<<i)	2473	0.0
(L+M())	7451	0.0	L.F.F[]	2437	0.0
(M()-i)	7181	0.0	(L-L.F)	2423	0.0
(L>>>i)	7181	0.0	-(L)	2423	0.0
(M() instanceof *Class*)	6305	0.0	(L<>l)	2322	0.0
(L<<i)	6013	0.0	(L&L)	2285	0.0
(L*i)	5818	0.0	((L.F>>i)&i)	2275	0.0
L[][]	5757	0.0	S.F	2214	0.0
(L.F-L.F)	5509	0.0	((char)L)	2201	0.0
((double)L)	5300	0.0	(S+i)	2132	0.0
(L*L)	5234	0.0	((float)L)	2129	0.0
L.F[][]	5230	0.0	((int)M())	2081	0.0

DOI: 10.1002/spe

Table XVIII. Abbreviations used in Table XVII.

`null`	≡	`ACONST_NULL`	`((`τ`)`α`)`	≡	typecast, τ is a primitive
`-(`α`)`	≡	negation			type or *Class*
(α+α)	≡	addition	`A`	≡	create new array
(α-α)	≡	subtraction	`S`	≡	static field
(α*α)	≡	mult	`F`	≡	non-static field
(α/α)	≡	div	`M()`	≡	method call
(α%α)	≡	mod/rem	(α `instanceof` κ)	≡	`instanceof`
(α&α)	≡	and	`"`	≡	string constant
(α|α)	≡	or	`f`	≡	`float` constant
(α^α)	≡	xor	`d`	≡	`double` constant
(α<<α)	≡	left shift	`l`	≡	`long` constant
(α>>α)	≡	signed right shift	`N`	≡	`NEW`
(α>>>α)	≡	`IUSHR` or `LUSHR`	(α<α)	≡	`DCMPL` or `FCMPL`
α[]	≡	array element	(α>α)	≡	`DCMPG` or `FCMPG`
α`.length`	≡	`ARRAYLENGTH`	(α<>α)	≡	`LCMP`
`i`	≡	`int` constant	`L`	≡	load local variable

static type of `o`, and the number of methods in `type(o)`'s subclasses that override `o`'s `M()`. A static class hierarchy analysis [12] is used to compute the receiver set.

The size of the receiver set has implications for, among other things, code optimization. A virtual method call that has only one member in its receiver set can be replaced with a direct call. Furthermore, if, for example, `o.M()`'s receiver set is {`Class1.M()`, `Class2.M()`}, then to expand `o.M()` inline, the code `if o instanceof Class1 then Class1.M() else Class2.M()` has to be generated. The larger the receiver set, the more type tests will have to be inserted.

To compute receiver sets for an `INVOKEVIRTUAL` instruction, we first resolve the method reference. We then gather all of the subclasses of the resolved method's parent class (including itself) and for each one look to see whether it contains a non-abstract method with the same name and signature as the resolved method. If so, we check to see whether the resolved method is accessible from the given subclass. If this is true, then the `INVOKEVIRTUAL` instruction could possibly execute the subclass' method, and it is added to the receiver set for the `INVOKEVIRTUAL` instruction.

For an `INVOKEINTERFACE` instruction, we perform the same test but we look instead at all implementors of the resolved method's parent interface. This set will contain all classes that directly implement the interface, as well as subclasses of those classes, and classes that implement any subinterfaces of the interface (i.e. anything that could be cast to the interface type). The `INVOKESPECIAL` and `INVOKESTATIC` instructions do not use dynamic method invocation; the method they will call can always be determined statically. Thus, they all have receiver sets of size 1.

Since we count only method bodies in the receiver sets, it is possible to have receiver sets of size 0. This can occur if an abstract class has no subclasses to implement its abstract methods, yet code is written to call its abstract methods with future subclasses in mind. Similarly, an `INVOKEINTERFACE` call may have no receivers if no classes implement the given interface.

DOI: 10.1002/spe

Table XIX. Common integer constants.

Most common `int` constants			Most common `long` constants		
Value	Count	%	Value	Count	%
0	634 484	20.5	0	19 617	29.2
1	611 382	19.7	1	3515	5.2
2	165 656	5.3	−1	2320	3.5
3	86 253	2.8	1000	1114	1.7
−1	78 187	2.5	287948901175001088	740	1.1
4	65 209	2.1	2	722	1.1
8	45 619	1.5	255	669	1.0
5	40 047	1.3	3	387	0.6
10	31 762	1.0	100	347	0.5
255	31 249	1.0	5	344	0.5
6	29 798	1.0	8388608	343	0.5
7	28 356	0.9	4294967295	331	0.5
9	25 497	0.8	10	323	0.5
16	24 931	0.8	7	304	0.5
32	19 401	0.6	71776119061217280	270	0.4
12	17 889	0.6	4	269	0.4
13	17 228	0.6	60000	217	0.3
11	15 763	0.5	9	199	0.3
15	15 008	0.5	541165879422	196	0.3
14	13 733	0.4	9223372036854775807	190	0.3
24	13 607	0.4	64	182	0.3
20	10 844	0.3	9007199254740992	170	0.3
48	9759	0.3	8	167	0.2
17	9513	0.3	-9223372036854775808	160	0.2
63	8963	0.3	500	159	0.2
46	8540	0.3	60	156	0.2
47	8214	0.3	36028797018963968	148	0.2
18	8115	0.3	2147483647	144	0.2
34	8029	0.3	67108864	139	0.2
31	7681	0.2	3600000	134	0.2
64	7602	0.2	17179869184	132	0.2
40	7187	0.2	144115188075855872	132	0.2
21	7044	0.2	140737488355328	130	0.2
100	6984	0.2	1024	129	0.2
45	6970	0.2	10000	125	0.2
23	6860	0.2	137438953504	123	0.2
19	6855	0.2	43980465111040	122	0.2
41	6631	0.2	1099511627776	122	0.2
30	6621	0.2	562949953421312	118	0.2
58	6551	0.2	17592186044416	118	0.2
128	6547	0.2	33554432	117	0.2
22	6510	0.2	268435456	117	0.2
25	6441	0.2	16384	109	0.2

DOI: 10.1002/spe

Table XX. Common real constants.

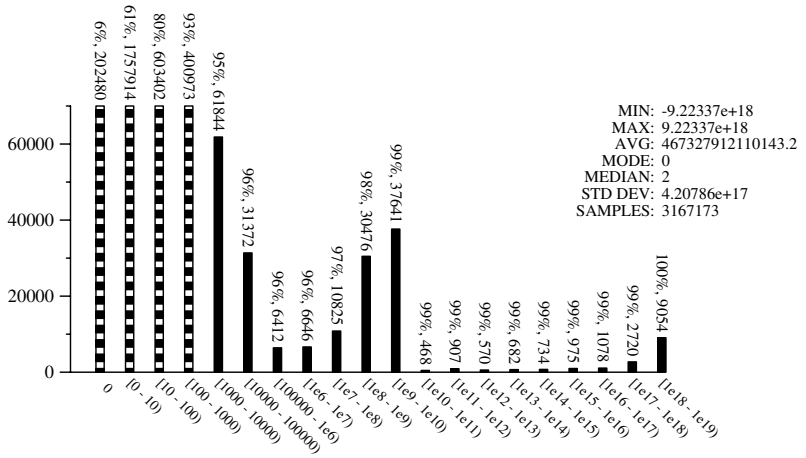
Most common `float` constants			Most common `double` constants		
Value	Count	%	Value	Count	%
`0.0`	4316	24.3	`0.0`	9295	25.0
`1.0`	2122	12.0	`1.0`	5146	13.8
`2.0`	839	4.7	`2.0`	1920	5.2
`0.5`	573	3.2	`0.5`	1296	3.5
`255.0`	319	1.8	`100.0`	710	1.9
`−1.0`	311	1.8	`10.0`	689	1.9
`4.0`	177	1.0	`5.0`	585	1.6
`100.0`	164	0.9	`-Infinity`	467	1.3
`10.0`	151	0.9	`−1.0`	463	1.2
`0.75`	146	0.8	`1000.0`	454	1.2
`64.0`	124	0.7	`3.0`	409	1.1
`3.0`	124	0.7	`3.141592653589793` (π)	364	1.0
`1000.0`	114	0.6	`NaN`	333	0.9
`20.0`	109	0.6	`4.0`	311	0.8
`90.0`	79	0.4	`0.25`	285	0.8
`3.1415927` (π)	73	0.4	`Infinity`	206	0.6
`NaN`	68	0.4	`8.0`	198	0.5
`57.29578` ($180/\pi$)	68	0.4	`1.797693···7E308` (*MAX*)	182	0.5
`50.0`	68	0.4	`180.0`	162	0.4
`6.2831855` (2π)	64	0.4	`1.5`	156	0.4
`6.0`	64	0.4	`0.1`	152	0.4
`3.4028235E38` (*MAX*)	62	0.3	`360.0`	145	0.4
`1.0E-4`	61	0.3	`20.0`	120	0.3
`180.0`	60	0.3	`6.283185307179586` (2π)	118	0.3
`5.0`	58	0.3	`−2.0`	112	0.3
`0.85`	58	0.3	`0.01`	107	0.3
`0.1`	58	0.3	`255.0`	105	0.3
`0.01`	52	0.3	`0.2`	104	0.3
`−10.0`	51	0.3	`6.0`	103	0.3
`0.0010`	47	0.3	`0.6`	92	0.2
`8.0`	45	0.3	`7.0`	91	0.2
`1.5`	45	0.3	`9.0`	88	0.2
`0.8`	45	0.3	`1.25`	83	0.2
`0.3`	45	0.3	`16.0`	77	0.2
`0.25`	45	0.3	`60.0`	75	0.2
`-Infinity`	44	0.2	`31.0`	72	0.2
`Infinity`	44	0.2	`26.0`	71	0.2
`100000.0`	42	0.2	`0.75`	71	0.2
`1.5707964` ($\pi/2$)	40	0.2	`12.0`	69	0.2
`0.70710677` ($1/\sqrt{2}$)	40	0.2	`0.05`	67	0.2
`−100.0`	38	0.2	`0.3`	66	0.2
`200.0`	37	0.2	`15.0`	65	0.2
`65536.0`	36	0.2	`645.0`	64	0.2

DOI: 10.1002/spe

Table XXI. Most common string constants.

Value	Count	%
empty string	36 456	3.5
" "	9003	0.9
newline	5281	0.5
")"	4860	0.5
"."	4718	0.5
"S"	4540	0.4
"'"	4201	0.4
"Q"	4139	0.4
":"	4083	0.4
","	3885	0.4
"R"	3796	0.4
"P"	3663	0.4
", "	3562	0.3
"/"	3481	0.3
"""	3113	0.3
"0"	3024	0.3
"name"	2725	0.3
"("	2561	0.2
"false"	2536	0.2
"true"	2461	0.2
"]"	2115	0.2
": "	2093	0.2
"Center"	1931	0.2
"-"	1699	0.2
"BC"	1658	0.2
"PvQ"	1649	0.2
">"	1634	0.2
"id"	1450	0.1
"P->Q"	1373	0.1
"P&Q"	1370	0.1
"java.lang.String"	1314	0.1
"line.separator"	1313	0.1
";"	1307	0.1
"W"	1237	0.1
"="	1210	0.1
"shortDescription"	1207	0.1
tab	1159	0.1
"}"	1151	0.1
"RvS"	1110	0.1
"null"	1107	0.1
"*"	1084	0.1
"A"	1074	0.1
"["	1046	0.1
"class"	1012	0.1
"~P"	996	0.1

DOI: 10.1002/spe



(a)

Value	Count	%
`0`	654101	20.7
`1`	695404	22.0
`2`	169760	5.4
$2^n, n > 1$	205877	6.5
$2^n - 1, n > 1$	198280	6.3
$2^n + 1, n > 1$	93544	3.0
other	1150207	36.3

(b)

Figure 34. Constant values: (a) distribution of integers (`int` and `long`); (b) integers (`int` and `long`) close to powers of two.

Figure 35(a) shows that 88% of all virtual method calls have a receiver set with size at most 2, with the average size being 4.5. It is interesting to note the large number of methods with a receiver size between 20 and 29. As can be expected, the average receiver set size is significantly larger for an interface method call. Figure 35(b) shows an average set size of 16.5.

7.6. Switches

Figure 36(a) measures the number of *case* labels for each `tableswitch` and `lookupswitch` instruction. We had to treat the `tableswitch` instruction specially, since it uses a contiguous range of label values. Not all of the labels in the `tableswitch` instruction necessarily appeared in the source code for the program. As a result, some of the branch targets for the cases will be the same as

DOI: 10.1002/spe

Table XXII. Most common calls to methods in the Java library.

Method	Count	%
java.lang.StringBuffer.append(String)StringBuffer	340 044	15.8
StringBuffer.toString()String	143 985	6.7
StringBuffer.<init>()void	93 837	4.3
Object.<init>()void	52 597	2.4
StringBuffer.<init>(String)void	48 408	2.2
String.equals(Object)boolean	46 645	2.2
java.util.Hashtable.put(Object,Object)Object	42 629	2.0
java.io.PrintStream.println(String)void	42 594	2.0
StringBuffer.append(int)StringBuffer	31 702	1.5
StringBuffer.append(Object)StringBuffer	27 284	1.3
String.length()int	25 505	1.2
String.valueOf(Object)String	20 146	0.9
java.lang.IllegalArgumentException.<init>(String)void	15 737	0.7
StringBuffer.append(char)StringBuffer	15 116	0.7
String.substring(int,int)String	14 441	0.7
java.util.Vector.size()int	13 817	0.6
java.util.Vector.addElement(Object)void	12 705	0.6
java.util.Vector.elementAt(int)Object	12 087	0.6
java.lang.System.arraycopy(Object,int,Object,int,int)void	11 969	0.6
java.util.Iterator.hasNext()boolean	11 891	0.6
String.charAt(int)char	11 831	0.5
java.util.Iterator.next()Object	11 800	0.5
java.lang.Integer.<init>(int)void	11 658	0.5
java.lang.Throwable.getMessage()String	11 216	0.5
java.util.Vector.<init>()void	10 434	0.5
java.util.List.add(Object)boolean	10 166	0.5
Object.getClass()java.lang.Class	9641	0.4
java.util.Hashtable.get(Object)Object	9584	0.4
String.equalsIgnoreCase(String)boolean	9391	0.4
java.util.List.size()int	9226	0.4
java.util.Map.put(Object,Object)Object	8830	0.4
java.util.List.get(int)Object	8797	0.4
java.lang.Class.forName(String)java.lang.Class	8641	0.4
java.util.Map.get(Object)Object	8313	0.4
java.awt.Container.add(java.awt.Component)java.awt.Component	8270	0.4
String.substring(int)String	7862	0.4
java.io.PrintWriter.println(String)void	7767	0.4
java.util.Enumeration.nextElement()Object	7539	0.3
java.lang.Class.getName()String	7288	0.3
String.startsWith(String)boolean	7186	0.3
String.indexOf(String)int	6960	0.3
java.util.ArrayList.<init>()void	6705	0.3
java.lang.Integer.parseInt(String)int	6667	0.3
java.util.Enumeration.hasMoreElements()boolean	6526	0.3
java.lang.NullPointerException.<init>(String)void	6403	0.3

DOI: 10.1002/spe

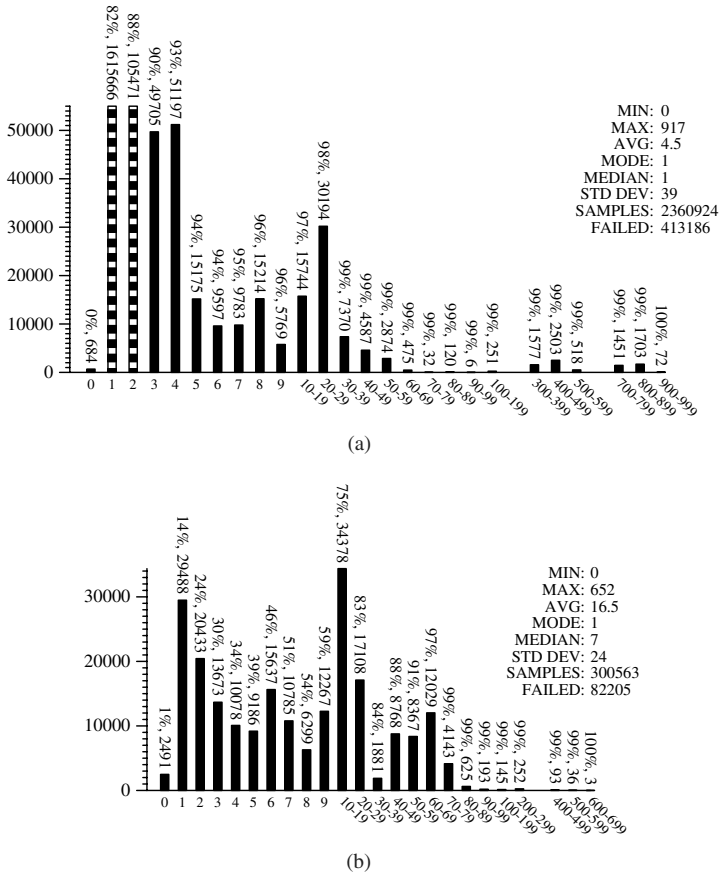


Figure 35. Number of receiver set sizes per (a) virtual method call (`invokevirtual`) and (b) interface method call (`invokeinterface`).

the *default* case target. Therefore, when computing the label set size and density of a `tableswitch` instruction, we ignore all of the labels whose branch targets are the same as the *default* case's target.

The figure shows that the average number of labels per switch is 12.8 and that 89% of the switches contain fewer than 30 labels.

Figure 36(b) shows the density of switch labels, computed as

$$\frac{number\_of\_case\_arms}{max\_label - min\_label + 1} \tag{1}$$

This measure is important for selecting the most appropriate implementation of switch statements [13,14]. In the JVM, the `tableswitch` instruction is used when the density is high and the `lookupswitch` is used when the density is low.

DOI: 10.1002/spe

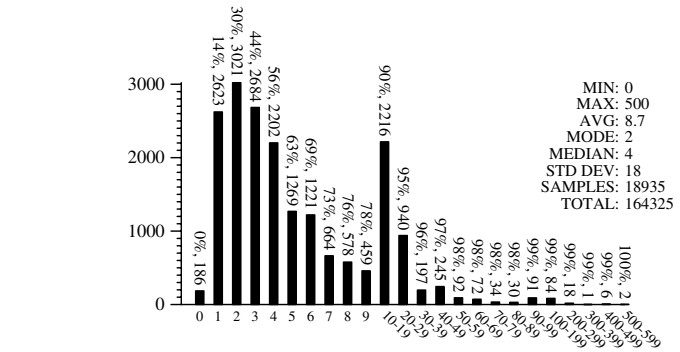


Figure 36. Switching statements: (a) number of case arms in `tableswitch` and `lookupswitch`; (b) label density of `tableswitch` and `lookupswitch`.

8. RELATED WORK

In a widely cited empirical study, Knuth conducted an analysis of 440 FORTRAN programs [1]. The study was conducted in an attempt to understand how FORTRAN was actually being used by typical programmers. By understanding how the language was being used, a better compiler could be designed. Each of the programs were subjected to static analysis in order to count common constructs such as assignment statements, ifs, gotos, do loops, etc. In addition, dynamic analysis was performed on 25 programs which examined the frequency of the constructs during a single execution of the program.

DOI: 10.1002/spe

The final analysis studied the effects of various local and global optimizations on the inner loops of 17 programs.

Knuth's study was the first attempt to understand how programmers actually wrote programs. Since that initial study, many similar explorations have been conducted for a variety of languages. Salvadori *et al.* [3] and Chevance and Heidet [2] both examined the profile of Cobol programs. Salvadori *et al.* looked at the static profile of 84 Cobol programs within an industrial environment. In addition to examining the frequency of specific constructs, they also studied the development history by recording the number of runs per day and the time interval between the runs. Chevance and Heidet studied the static nature of Cobol programs through the number of occurrences of source-level constructs in more than 50 programs. The authors took their study a step further by computing the frequency of the constructs as the program executed. In this study, for categories of data were examined: constants, variables, expressions, and statements.

Other than Chevance and Heidet [2], most studies of programmer behavior have concentrated on the static structure of programs. Of equal importance is to examine how programs change over time. Collberg *et al.* [15] showed how to visualize the evolution of a program by taking snapshots of its development from a CVS repository and presenting these data using a temporal graph-drawing system.

Cook and Lee [4] undertook a static analysis of 264 Pascal programs to gain an understanding of how the language was being used. The analysis was conducted within 12 different contexts, e.g. procedures, then-parts, else-parts, for-loops, etc. In addition, they compared their results with those of other language studies. Cook [16] conducted a static analysis of the instructions used in the system software on the Lilith computer. An analysis of APL programs was conducted by Saal and Weiss [5,6].

Antonioli and Pilz [17] conducted the first analysis of the Java class file. The goal of their study was to answer three questions. (1) What is the size of a typical class file? (2) How is the size of the class file distributed between its different parts? (3) How are the bytecode instructions used? To answer these questions, they examined six programs with a total of 4016 unique classes. In contrast to the present study, they examined the size in bytes of each of the five parts of a class file (i.e. header, constant, class, field, and method). They also examined instruction frequencies to see what percentage of the instruction set was actually being used. They found that on average only 25% of the instruction set was used by any one program. Our analysis does not focus on the frequency of a particular instruction per program but instead looks at the frequency over all programs. Overall, their study is different from ours in that they were interested in answering a few very specific questions, where our analysis is focused on obtaining a complete understanding of JVM programs.

Gustedt *et al.* [18] conducted a study of Java programs that measures the tree width of CFGs. The tree width is effected by such constructs as `goto` usage, short-circuit evaluation, multiple exits, `break` statements, `continue` statements, and returns. The authors examined both Java API packages as well as Java applications obtained through Internet searches.

O'Donoghue *et al.* [19] performed an analysis of Java bytecode bigrams. Their analysis was performed on 12 benchmark applications. The only similarity between their data and ours is that we both found `aload_0, getfield` to be the most frequently occurring bigram. We attribute the differences to the small sample size used in their study.

One of the byproducts of our analysis is a large repository of publicly available data on Java programs. Appel [20] maintains a collection of interference graphs which can be used in studying graph-coloring algorithms. The availability of such repositories is highly useful in the study of compiler implementation techniques.

DOI: 10.1002/spe

9. DISCUSSION AND SUMMARY

In this paper we have performed a static analysis of 1132 Java programs obtained from the Internet. Through the use of SandMark, we were able to analyze the structure of the Java bytecode. Our analysis ranged from simple counts, such as methods per class, instructions per method, and instructions per basic block, to structural metrics such as the complexity of CFGs.

Our main goal in conducting the study was to use the data in our research on software protection, however we believe these data are useful in a variety of settings. These data could be used in the design of future programming languages and virtual machine instruction sets, as well as in the efficient implementation of compilers.

It would be interesting to perform a similar study of Java source code. Even though Java bytecode contains much of the same information as in the source from which it was compiled, some aspects of the original code are lost. Examples include comments, source code layout, some control structures (when translated to bytecode, `for` and `while` loops may be indistinguishable), some type information (Booleans are compiled to JVM integers), etc.

Owing to our random sampling of code from the Internet, it is possible that our set of Java jar-files is somewhat skewed. It would be interesting to further validate our results by comparing against a different set of programs, such as standard benchmark programs (for example, SpecJVM [21]), or programs collected from standard source code repositories (for example, `sourceforge.net`).

We would also welcome studies for other languages. It would be interesting to validate our results by performing a similar study for MSIL, the bytecode generated from C# programs, since MSIL and JVM (and C# and Java) share many common features. It would also be interesting to compare our results with languages very different from Java, such as functional, logic, and procedural languages. It might then be possible to derive a set of 'linguistic universals', programming behaviors that apply across a range of languages. Such information would be invaluable in the design of future programming languages.

Our experimental data and the SandMark tool that was used to collect it can be downloaded from http://sandmark.cs.arizona.edu/download.html.

REFERENCES

1. Knuth DE. An empirical study of FORTRAN programs. *Software—Practice and Experience* 1971; **1**:105–133.
2. Chevance RJ, Heidet T. Static profile and dynamic behavior of COBOL programs. *SIGPLAN Notices* 1978; **13**(4):44–57.
3. Salvadori A, Gordon J, Capstick C. Static profile of COBOL programs. *SIGPLAN Notices* 1975; **10**(8):20–33.
4. Cook RP, Lee I. A contextual analysis of Pascal programs. *Software—Practice and Experience* 1982; **12**:195–203.
5. Saal HJ, Weiss Z. Some properties of APL programs. *Proceedings of 7th International Conference on APL*. ACM Press: New York, 1975; 292–297.
6. Saal HJ, Weiss Z. An empirical study of APL programs. *International Journal of Computer Languages* 1977; **2**(3):47–59.
7. Stata R, Abadi M. A type system for Java bytecode subroutines. *Conference Record of POPL 98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Diego, CA, 1998. ACM Press: New York, 1998; 149–160.
8. Collberg CS, Tomborson C. Watermarking, tamper-proofing, and obfuscation—tools for software protection. *IEEE Transactions on Software Engineering* 2002; **8**(8):735–746.
9. Collberg C, Myles M, Huntwork A. SANDMARK—A tool for software protection research. *IEEE Magazine of Security and Privacy* 2003; **1**(4):40–49.
10. Lindholm T, Yellin F. *The Java Virtual Machine Specification* (2nd edn). Addison-Wesley: Reading, MA, 1999.

DOI: 10.1002/spe

11. Cousot P, Cousot R. An abstract interpretation-based framework for software watermarking. *Proceedings of the ACM Conference on Principles of Programming Languages*. ACM Press: New York, 2004.
12. Dean J, Grove D, Chambers C. Optimization of object-oriented programs using static class hierarchy analysis. *Proceedings of the 9th European Conference on Object-Oriented Programming*. Springer: Berlin, 1995; 77–101.
13. Bernstein R. Producing good code for the case statement. *Software—Practice and Experience* 1985; **15**(10):1021–1024.
14. Kannan S, Proebsting TA. Correction to 'producing good code for the case statement'. *Software—Practice and Experience* 1994; **24**(2):233.
15. Collberg C, Kobourov S, Nagra J, Pitts J, Wampler K. A system for graph-based visualization of the evolution of software. *Proceedings of the ACM Symposium on Software Visualization*, June 2003. ACM Press: New York, 2003.
16. Cook RP. An empirical analysis of the Lilith instruction set. *IEEE Transactions on Computers* 1989; **38**(1):156–158.
17. Antonioli DN, Pilz M. Analysis of the Java class file format. *Technical Report*, University of Zurich, 1998. Available at: ftp://ftp.ifi.unizh.ch/pub/techreports/TR-98/ifi-98.04.ps.gz.
18. Gustedt J, Mæhle OA, Telle JA. The treewidth of Java programs. *Proceedings of the 4th International Workshop on Algorithm Engineering and Experiments (ALENEX)* (*Lecture Notes in Computer Science*, vol. 2409). Springer: Berlin, 2002.
19. O'Donoghue D, Leddy A, Power J, Waldron J. Bigram analysis of Java bytecode sequences. *Proceedings of the 2nd Workshop on Intermediate Representation Engineering for the Java Virtual Machine*. National University of Ireland: Ireland, 2002; 187–192.
20. Appel A. Sample graph coloring problems. http://www.cs.princeton.edu/~appel/graphdata/.
21. SpecJVM98. http://www.specbench.org/osg/jvm98.

DOI: 10.1002/spe