Object Oriented Programs and a Stack Based Virtual Machine

JT Waldron

School of Computer Applications, Dublin City University, Dublin 9, Ireland.
On sabbatical July 1999 - July 2000 at School of Mathematics, Statistics & Information Systems, University of Natal, Pietermaritzburg 3209, South Africa,
jwaldron@compapp.dcu.ie

Abstract

Dynamic quantitative measurements of Bytecode and Stack Frame Usage by Eiffel and Java Programs in the Java Virtual Machine are made. Two Eiffel programs are dynamically analysed while executing on the JVM, and the results compared with those from the Java Programs. The aim is to examine whether properties like instruction usage and stack frame size are properties of the Java programming language itself or are exhibited by Eiffel programs as well. Investigations analyse how the different assertion checking and optimizations possible using the SmallEiffel compiler affect bytecode and stack frame usage. Remarkably local_load, push_const and local_store instruction categories always account for very close to 40% of instructions executed, a property of the Java Virtual Machine for both the Java and Eiffel programming languages, irrespective of compiler or compiler optimizations used. Java programs executed 75% of their bytecodes within the API suggesting a way to improve the speed of Java programs would be to compile the API methods to native instructions and save these on disk in a standard format, cutting the time spent interpreting programs. Only 4.8% of instructions were in the API when Eiffel programs executed.

Keywords: *Virtual Machines, Languages and compilers, Interpreters, Run-time environments*
Computing Review Categories: *B.1.4 D.3.4*

1 Introduction

In this paper two Eiffel [12] programs are dynamically analysed while executing on the Java Virtual Machine, and the results compared with those from five Java Programs. The aim is to examine whether properties like instruction usage and stack frame size are properties of the Java programming language itself or are exhibited by Eiffel programs as well and therefore are innate properties of the stack based virtual machine. The long term aim of this research is to develope principles of virtual machine design to support platform independent efficient execution of object oriented languages like Eiffel as well as Java.

This paper shows that, for the Java programs studied, dynamic bytecode usages were almost identical when the same program was compiled by different Java compilers. In addition the -O option only had small effects on bytecode usage patterns. Investigations examine how the different assertion checking and optimizations possible using the SmallEiffel compiler affect bytecode and stack frame usage in the Java Virtual Machine.

In Section 2 we discuss the Java Virtual Machine (JVM) bytecodes. Section 3 describes the suite of programs chosen to investigate usage of the bytecodes by real programs. Section 4 gives the bytecode frequencies over the Java and Eiffel programs and looks at the dynamic bytecode usages produced using the different compiler optimizations possible with SmallEiffel. Section 5 looks at dynamic stack frame usage by the Java and Eiffel programs studied and a discussion of the results is given in Section 6.

2 Java Virtual Machine Bytecode Frequency

The dynamic frequency of an instruction is the number of times it is executed during a program run. The static bytecode frequency, which is the number of times a bytecode appears in a class file or program has been studied in [4] where a wide difference was found between the bytecodes appearing in different class files, and each class file used on average 25 different bytecodes.

Java originated in 1991 as the OAK project aiming to produce a software development environment for small distributed embedded systems. The design involves compiling to a machine independent instruction set similar to UCSD Pascal [6]. The architecture is a stack machine with almost no addressing modes, a highly regular instruction set, and very dense instruction encoding (1.9 bytes per instruction for the programs studied in [4]). The instruction set of the Java Virtual Machine is fully documented in [11]. The Java stack is divided into frames and there is one frame per method invocation which has its own local registers. The Java instruction set contains an unusual amount of type information, and there are restrictions on the use of the operand stack, so that at every code point each slot in the stack and each local variable has a type. The type can be determined statically by a bytecode verifier, which means that stack underflow and overflow, the types of all parameters to instructions and object field references, and potential illegal conversions can be checked statically. The

type information in the bytecode representation should also help a just in time (JIT) compiler in translating programs into native machine code [10].

The simplest way the Virtual Machine executes a Java program is by interpreting the bytecodes. Performance can be improved by using a JIT compiler to produce native machine code at run time. In order to achieve a performance competitive with the code produced by a native compiler however, it would be necessary to apply the same code-generation techniques, which seems too expensive for JIT compilation. Even though the Java platform dynamically links and loads classes as they are needed, most execution time is spent in a small portion of the code [5] [3]. The Hotspot Virtual Machine design uses dynamic compilation to only convert the most heavily used parts of a program to native machine instructions, and interprets the bulk of the code, avoiding the overhead of compiling code that is infrequently or never executed.

In order to study dynamic bytecode usage it was necessary to modify the source code of a Java Virtual Machine. Kaffe [15] is an independent implementation of the Java Virtual Machine which was written from scratch and is free from all third party royalties and license restrictions. It comes with its own standard class libraries, including Beans and Abstract Window Toolkit (AWT), native libraries, and a highly configurable virtual machine with a JIT compiler for enhanced performance. Kaffe is available under the Open Source Initiative and comes with complete source code, distributed under the GNU Public License.

3 Programs Measured

Programs written in both Java and Eiffel were studied to see if bytecode and stack frame usages were a property of the Java language itself, or a property of the Virtual Machine.

3.1 Java Programs

A selection of different Java programs was studied to compare the way different Java applets and applications use the bytecodes. The 'atomic' applet symbolises nuclear forces at work. It is a simple applet showing an animated atom with whizzing electrons. Lots of configuration options make it easy to customise. The fireworks applet displays a number of fireworks rockets and animates them according to a set of user definable attributes.

Jasmin [13] is a Java Assembler Interface. It takes ASCII descriptions for Java classes, written in a simple assembler-like syntax using the Java Virtual Machine instruction set. It converts them into binary Java class files suitable for loading into a Java Virtual Machine implementation. Jasmin itself is written in Java. Jasmin was measured assembling Count.j one of the example programs distributed with it.

Java Compiler Compiler (JavaCC) [2] is currently the most popular parser generator for use with Java applications. A parser generator is a tool that reads a grammar specification and converts it to a Java program that can recognise matches to the grammar. In addition to the parser generator itself, JavaCC provides other standard capabilities related to parser generation such as tree building actions, debugging, etc. JJTree, the tree builder, has now become popular and many users have started using JJTree along with JavaCC. The bytecodes JavaCC used while running javacc SPL.jj were measured. SPL (Stupid Programming Language) is one of the example programs distributed with JavaCC. JJTree was also measured while running jjtree SPL.jjt.

3.2 Eiffel Programs

SmallEiffel is intended to be a complete, small, very fast and free Eiffel compiler, available for a wide range of platforms. It includes an Eiffel to C compiler, an Eiffel to Java bytecode compiler, a documentation tool, and a pretty printer. SmallEiffel uses an innovative strategy involving whole system analysis which allows compilation to be often faster than the incremental compilation of traditional compilers [8]. It was originally designed at the LORIA lab, Nancy, France, in 1994-95, and has been used worldwide by many individuals and Universities since September 1995. Version -0.78, released on Saturday June 5th, 1999 was used for these comparisons. As it is not yet possible to create applets with compile_to_jvm, no Eiffel applets were studied.

The main concern of the Gobo Eiffel Project [1] is to provide the Eiffel community with free and portable Eiffel tools and libraries for use by any Eiffel compiler. Two utilities from Gobo Eiffel 1.5 were measured — Gobo Eiffel Lex, version 1.5 (gelex) and Gobo Eiffel Yacc, version 1.5 (geyacc). Gobo Eiffel Lex is a tool for generating Eiffel programs that perform pattern-matching on text. Gelex reads a given input file for the description of the scanner to be generated. The description is in the form of pairs of regular expressions and Eiffel code, called rules. Gelex generates as output an Eiffel class equipped with routines to analyze input text for occurrences of the regular expressions. Whenever one is found, the corresponding Eiffel code is executed.

Gobo Eiffel Yacc [1] is a general-purpose parser generator that converts a grammar description for an LALR(1) context-free grammar into an Eiffel class equipped with routines to parse that grammar. Geyacc may be used to develop a wide range of language parsers, from those used in simple desk calculators to complex programming languages. Geyacc is very similar to yacc and GNU bison. Anyone familiar with these utilities and fluent in Eiffel programming should be able to use geyacc with little trouble.

Gelex was measured running *gelex ascii2ps.l*, a simple ASCII to PostScript filter which can be used as a pretty printer. Geyacc was measured running *geyacc -o calc_parser.e calc_parser.y* a simple calculator.

	API		
	non-native method %	bytecodes %	bytecodes per method
atomic	43.5	79.8	37.7
fire works	72.8	64.9	24.1
jasmin	86.4	89.7	22.4
JJTree	61.6	63.9	15.5
JavaCC	71.3	72.4	19.1
average	67.1	74.1	23.8
	non-API		
	non-native method %	bytecodes %	bytecodes per method
atomic	56.5	20.2	7.3
fire works	27.2	35.1	34.8
jasmin	13.6	10.3	16.4
JJTree	38.4	36.1	14.0
JavaCC	28.7	27.6	18.2
average	32.9	25.9	18.1

Table 1: Comparison of dynamic percentages of instructions and non-native method calls made by methods in the API and in the program classes for the Java programs studied.

4 Instruction Execution Frequencies

4.1 Java Programs

In order to fully study the dynamic execution of bytecodes, those used within API methods were measured separately from those in the program's methods. Table 1 compares the dynamic percentages of instructions and non-native method calls in the API and in the program's classes for the Java programs studied. It can be seen that 74.1% of bytecodes executed and 67.1% of Java methods are in the API. *java/lang*, *java/awt* and *java/util* contain the API methods most frequently executed by the programs. Table 1 implies that the behaviour of these methods will dominate the overall behaviour of a program. One obvious suggestion to improve the speed of Java programs would be to compile the API methods to native instructions and save these on disk in a standard format, cutting the time spent interpreting, which should give a dramatic increase in performance. Table 1 shows that on average each Java method in the programs (i.e. not in the API) only used 18.1 bytecodes when executed and that Java API methods executed on average 23.8 bytecodes.

Total (API and non-API) dynamic bytecode execution frequencies of each instruction are given in Table 2 for the atomic, fireworks, jasmin, JavaCC, and JJTree Java programs. API and non-API methods use similar bytecodes. The unabridged version of the instruction frequencies given in Table 2 is available at [14]. The last column in Table 2 shows the average of the five benchmarks

$$f_i = \frac{1}{5}\sum_{k=1}^{5}\frac{100 \times c_{ik}}{\sum_{i=1}^{256} c_{ik}}$$

where c_{ik} is the number of times bytecode i is executed during the execution of program k. f_i is an approximation of that bytecode's usage for a typical program. The Java bytecodes selected for inclusion in this Table 2 were the 35 most commonly used bytecodes. For the purposes of this

Instruction	atom	fire	jas	jcc	jjt	f_i
aload_0	7.9	21.3	15.7	11.3	11.9	13.6
getfield	4.8	20.3	13.2	9.7	9.4	11.5
iload_1	3.8	2.3	4.5	3.9	2.9	3.5
invokevirtual	3.6	2.9	3.5	2.7	3.2	3.2
iload_2	3.6	3.7	2.8	3.3	2.5	3.2
aload_1	1.0	4.6	1.8	3.3	4.6	3.1
iload	6.3	1.0	2.8	2.6	1.5	2.8
iload_3	1.7	2.3	2.9	3.2	3.2	2.7
getstatic	0.1	0.1	0.1	5.5	6.5	2.5
invokestatic	6.1	1.5	0.3	2.0	2.7	2.5
iadd	2.1	1.4	3.6	2.8	2.4	2.5
dmul	9.9	0.7	0.0	0.0	0.0	2.1
ireturn	3.0	1.2	2.0	1.8	2.1	2.0
return	0.8	2.3	1.4	2.5	3.0	2.0
putfield	0.5	4.1	2.3	1.5	1.8	2.0
iconst_0	0.4	0.7	1.8	2.9	3.1	1.8
iconst_1	0.2	1.9	2.3	2.3	2.2	1.8
dup	0.2	2.9	1.9	1.7	1.9	1.7
goto	0.8	0.5	2.5	2.0	1.6	1.5
if_icmpge	1.7	0.8	2.4	1.4	0.8	1.4
bipush	1.1	0.5	2.0	1.8	1.8	1.4
iinc	0.7	0.5	2.1	1.6	1.0	1.2
ldc2w	5.5	0.1	0.0	0.1	0.1	1.2
i2d	4.9	0.6	0.0	0.0	0.0	1.1
if_icmpne	0.2	0.7	1.6	1.8	1.0	1.1
ifeq	0.2	0.4	0.6	1.7	1.9	1.0
if_icmplt	0.1	1.2	0.9	1.5	1.5	1.0
areturn	0.1	0.2	1.3	1.0	1.7	0.9
invokespecial	0.3	0.6	1.0	1.2	1.6	0.9
aload	0.4	0.2	2.3	1.0	0.7	0.9
iload_0	1.9	0.3	0.1	1.0	1.2	0.9
caload	0.3	0.3	2.1	1.2	0.7	0.9
isub	1.9	0.1	0.5	0.6	0.9	0.8
d2i	2.7	1.3	0.0	0.0	0.0	0.8
istore_3	0.2	1.1	1.5	0.5	0.5	0.8

Table 2: Total (API and non-API) dynamic bytecode execution frequencies as percentages for the Java programs studied. The last column shows the average of the five benchmarks.

Category	Opcode	Bytecodes
misc	5	nop,iinc,athrow,wide,breakpoint
push_const	20	1-20
local_load	25	21-45
array_load	8	46-53
local_store	25	54-78
array_store	8	79-86
stack	9	87-95
arithmetic	24	96-119
logical_shift	6	120-125
logical_boolean	6	126-131
conversion	15	133-147
comparison	5	148-152
conditional_branch	16	153-166,198,199
unconditional_branch	2	goto,goto_w
subroutine	3	jsr,ret,jsr_w
table_jump	2	tableswitch,lookupswitch
method_return	6	172-177
object_fields	4	178-181
method_invoke	4	182-185
object_manage	3	new,checkcast,instanceof
array_manage	4	188-190,197
monitor	2	monitorenter,monitorexit

Table 3: Categories of Java bytecodes.

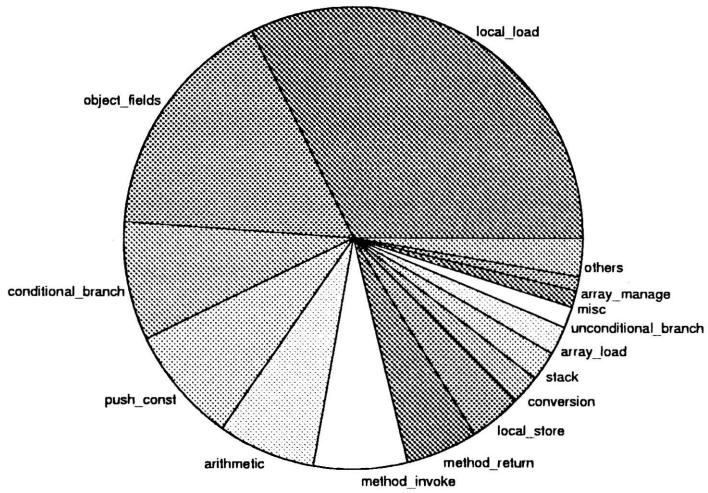

Figure 1: Total (API and Non-API) dynamic bytecode execution frequencies by category by the Java programs studied.

study, the 202 bytecodes can be split into 22 categories as shown in Table 3. By assigning those instructions that behave similarly into groups it is possible to describe clearly what is happening. Table 2 is summarised in Figure 1.

4.2 Bytecodes Produced by Different Compilers

An interesting question is to what extent the choice of compiler influenced the bytecodes executed by the Java Programs under scrutiny. Source code was not available for JavaCC or JJTree, so it was not possible to fully investigate this. Table 4 gives non-API dynamic bytecode execution frequencies for the Jasmin assembler application compiled by Borland's Jbuilder2, Microsoft Visual J++ version 6.0 and and Sun's JDK 1.1.7 compilers. Bytecodes executed in API methods were excluded from Table 4 as the API libraries were not recompiled. Remarkably similar frequencies were obtained using the different compilers.

An interesting problem was posed when trying to look at the bytecodes used when the atomic applet was compiled by different compilers since the applet is an infinite loop that runs until exited, making it impossible to reproduce executions exactly. One solution would be to stop the virtual machine after a certain number of instructions, but this would be unfair if one complier was more efficient and generated fewer bytecodes, so the source code of the applet was altered to rotate the atom a precise number of times and then exit. Table 5 shows non API dynamic bytecode execution frequencies for the 'atomic' applet compiled with Borland's Jbuilder2, Microsoft Visual J++ version 1.0, Sun's JDK 1.1.7 and JDK 1.2, and Pizza. The bytecodes executed were almost identical in all cases, again implying the main difference between Java compilers may be the speed of compilation.

Cream [7] is an optimizer for Java bytecodes that uses dead code elimination, loop invariant removal as well as several variations of side effect analysis. The similarity found here between bytecode usage by different compilers may be because many traditional optimisations performed by compilers cannot be used when converting a Java program to bytecode. For example, there are no choices about registers to use, and it is very difficult to optimise method calls. According to [3] most method invocations in the Java programming language are virtual (potentially polymorphic), which happens much less frequently in C++ where virtual calls are not the default.

If most method invocations are assumed to be virtual, method invocation performance is more dominant, and static compiler optimisations (especially global optimisations like inlining) are much harder to perform. Traditional optimisations are often most effective between calls, and the decreased distance between calls in the Java programming language can significantly reduce the effectiveness of such optimisations, since they have smaller sections of code to work with. Java programs can change on-the-fly due to the ability to perform dynamic loading of classes, making it far more difficult to perform many types of global optimisation, since the compiler must not only be able to detect when these optimisations become invalid due to dynamic loading, but also must be able to undo and/or redo those optimisations during program execution, even if they involve active methods on the stack, without compromising or impacting Java program execution semantics in any way.

Other difficulties arise in optimisation because the Java programming language is dynamically safe [11], meaning programs are guaranteeded not to violate the language semantics, or directly access unstructured memory. This means dynamic type-tests must frequently be performed when casting, and when storing into object arrays. The Java programming language allocates all objects on the heap, whereas in C++ many objects are stack allocated allowing potential for optimisation. This means that object allocation rates are much higher for the Java programming language than for C++. In addition, because the Java programming language is garbage-collected, it has very different types of memory allocation overhead including po-

Instruction	Jbuilder2	j++ 6.0	JDK117	JDK117 -O	f_i
aload_0	10.6	11.0	11.0	11.2	10.9
iload	7.2	7.4	7.4	7.7	7.4
getfield	5.7	5.8	5.8	6.1	5.8
invokevirtual	5.2	5.4	5.4	4.3	5.1
dup	5.0	4.9	4.9	5.1	5.0
aload	4.8	4.9	4.9	5.1	4.9
bipush	4.4	4.4	4.4	4.6	4.5
iload_1	3.2	3.3	3.3	3.5	3.3
saload	2.9	3.0	3.0	3.1	3.0
sastore	3.1	2.9	2.9	3.0	3.0
iconst_2	2.8	2.9	2.9	3.0	2.9
iconst_1	2.5	2.5	2.5	2.7	2.5
invokespecial	2.3	2.4	2.4	2.5	2.4
iload_2	2.2	2.3	2.3	2.4	2.3
istore	2.0	2.0	2.0	2.1	2.0
iinc	1.7	1.8	1.8	1.9	1.8
iload_3	1.7	1.8	1.8	1.9	1.8
iconst_m1	1.8	1.8	1.8	1.9	1.8
iconst_0	2.5	1.6	1.6	1.6	1.8
arraylength	1.6	1.7	1.7	1.7	1.7
imul	1.3	1.3	1.3	1.4	1.3
if_icmpne	1.2	1.3	1.3	1.3	1.3
iadd	1.3	1.3	1.3	1.4	1.3
goto	1.6	1.1	1.1	1.2	1.2
istore_3	1.2	1.2	1.2	1.3	1.2
if_icmpeq	1.1	1.1	1.1	1.2	1.1
if_icmple	0.8	1.1	1.1	1.2	1.1
aload_2	1.0	1.0	1.0	1.0	1.0
aastore	1.0	1.0	1.0	1.0	1.0
isub	0.8	0.9	0.9	0.9	0.9
if_icmplt	0.0	1.2	1.2	1.2	0.9
ldc1	1.5	0.7	1.3	0.1	0.9
iload_0	1.1	1.2	1.2	0.0	0.9
idiv	0.8	0.8	0.8	0.9	0.8
if_icmpge	1.8	0.4	0.4	0.4	0.8
Total	65608	63774	63771	61218	

Table 4: Non-API dynamic bytecode execution frequencies for the Jasmin assembler application using different compilers.

Instruction	Jbuilder2	j++ 1.00	JDK117	JDK12	pizza	f_i
invokestatic	13.4	13.4	13.4	13.5	13.4	13.4
aload_0	9.6	9.6	9.6	9.6	9.6	9.6
getfield	9.4	9.4	9.4	9.4	9.4	9.4
iconst_2	9.4	9.4	9.4	9.4	9.4	9.4
isub	9.3	9.3	9.3	9.3	9.3	9.3
dmul	8.8	8.8	8.8	8.8	8.8	8.8
i2d	8.8	8.8	8.8	8.8	8.8	8.8
ldc2w	8.8	8.8	8.8	8.8	8.8	8.8
iload_1	8.8	8.8	8.8	8.8	8.8	8.8
dload_3	8.8	8.8	8.8	8.8	8.8	8.8
ineg	4.4	4.4	4.4	4.4	4.4	4.4
invokevirtual	0.2	0.2	0.2	0.2	0.2	0.2
dstore	0.1	0.1	0.1	0.1	0.1	0.1
nop	0.0	0.0	0.0	0.0	0.0	0.0
lsub	0.0	0.0	0.0	0.0	0.0	0.0
fsub	0.0	0.0	0.0	0.0	0.0	0.0

Table 5: Non-API dynamic bytecode execution frequencies for the 'atomic' applet for 5 different compilers.

tentially scavenging and write-barrier overhead than C++.

The -O directs the JDK 1.1.7 compiler to try to generate faster code by inlining static, final and private methods. This option may slow down compilation, make larger class files, and/or make it difficult to debug. This option informs the compiler that all generated class files are guaranteed to be delivered and upgraded as a unit, enabling optimisations that may otherwise break binary compatibility, so this option needs to be used with discretion. The Java Language Specification [10] section 13.4.21 describes situations in which it is legal to use a Java compiler to inline methods. The compiler will only optimise code for which source is available during the compilation, so the only *.java* files discoverable by the compiler should be for classes intended to be delivered or upgraded as a unit. Table 4 shows a decrease in use of invokevirtual and a decrease in overall bytecode use for Jasmin compiled with the -O option. In the most recent Java systems (Java 1.2), the inlining abilities of *javac* have been vastly reduced. This was necessary because the previous inlining was too aggressive; it would sometimes produce class files that failed verification.

4.3 Bytecode Usage by Eiffel Programs

The Eiffel programming language supports the concept of "design by contract", where the operation of methods can be predicated by using various forms of assertions. These assertions, including the checking of pre- and post-conditions as well as loop and class invariants, forms an important part of the design of Eiffel programs and has a corresponding impact on the generated Java bytecodes. The SmallEiffel compiler supports 8 modes of compilation, allowing a fine-grained degree of control over assertion checking.

-boost Compilation mode with the highest degree of optimization. There is no target's existence test, no system-level validity checking. Some routines are inlined. No code is generated to get an execution trace in case of failure. No assertion is checked.

-no_check Compilation mode in which no Eiffel assertion is checked. The target's existence test is performed. Some code is generated for the system-level validity checking, and to produce an execution trace (an execution stack is managed). There is no inlining and no assertion check.

-require_check Compilation mode in which Eiffel preconditions are checked. The generated code is similar to the previous one, but also includes code to test preconditions (require).

-ensure_check The generated code is similar to the previous one, but also includes code to test postconditions (ensure).

-invariant_check The generated code is similar to the previous one, but also includes code to test class invariants.

-loop_check The generated code is similar to the previous one, but also includes code to test loop variants and loop invariants.

-all_check The default mode. The generated code is similar to the previous one, but also includes code for the check instruction.

-debug_check The generated code is similar to the previous one, but also includes code for debug instructions. All debugs are checked regardless of the optional string key.

Eiffel programs were measured compiled under two different modes, firstly -all_check and secondly with the -no_check option, a compilation mode in which no Eiffel assertion is checked. Of equal significance, this mode also disables the inlining optimisations of the SmallEiffel compiler (as described in [16]).

The first difference noted between the Eiffel programs and the Java Programs is the amount of execution time spent in the non-API program methods. This was 98.1% of instructions for geyacc and 99.2% for gelex when these were compiled with all checks (87.2 and 96.3 respectively with no checks). This result is not surprising since many low level routines are written in pure Eiffel. When gelex is compiled with all checks, it takes 1.3×10^8 total instructions and 4.5×10^6 methods. With the no checks option this falls to 3.3×10^7 instructions and 1.2×10^6 methods (all quantities include API methods).

Total (API and non-API) dynamic bytecode execution frequencies for the Eiffel programs studied when compiled using SmallEiffel no_check and all_check modes are shown in Table 6. Total (API and Non-API) dynamic bytecode execution frequencies by the categories in Table 3 by the Eiffel programs studied when compiled using SmallEiffel no_check modes are shown in Figure 2. One immediately noticeable difference between the Java programs and the Eiffel programs is the greater use of the `iconst_1` and `ifne` instructions in the latter. This is a direct consequence of the "design by contract" approach, and the resultant assertion checking during the execution of the Eiffel programs. For example, an object's integrity is represented by a `check_flag` field, and this is checked by comparison with the value 1 (pushed using the `iconst_1` instruction), with a resulting branch if this test fails (using an `ifne` instruction). Total (API and Non-API) dynamic bytecode execution frequencies by the categories in Table 3 by the Eiffel programs studied when compiled using SmallEiffel no_check mode, a compilation mode in which no Eiffel assertion is checked, are shown in Figure 3. The results are remarkably similar to those for Java Programs (Figure 1) suggesting that the instruction set usage is a property of the Java Virtual Machine design, rather than the programming language used.

no_check mode				all_check mode			
instruction	geyacc	gelex	f_i	instruction	geyacc	gelex	f_i
aload_0	14.4	10.8	12.6	iconst_1	9.8	13.0	11.4
getfield	5.6	9.3	7.5	aload_0	8.6	10.2	9.4
checkcast	5.6	9.3	7.5	ifne	7.9	6.6	7.2
instanceof	9.5	3.4	6.5	getfield	4.8	8.1	6.4
dup	5.1	6.3	5.7	iconst_0	4.9	6.0	5.5
ifeq	8.2	1.8	5.0	ifeq	5.0	4.8	4.9
iconst_0	3.9	4.8	4.3	checkcast	5.6	4.2	4.9
ifne	3.6	4.3	4.0	goto	3.4	4.4	3.9
istore_2	2.3	5.6	3.9	iload_1	2.8	4.4	3.6
goto	3.5	3.5	3.5	dup	4.8	2.5	3.6
iload_1	2.7	4.2	3.5	getstatic	3.8	2.6	3.2
invokevirtual	2.6	3.4	3.0	invokevirtual	3.6	2.6	3.1
iload_2	2.3	3.3	2.8	putstatic	1.3	3.4	2.4
ifnull	2.2	3.0	2.6	instanceof	3.8	1.0	2.4
iconst_1	2.4	2.8	2.6	ireturn	2.0	2.6	2.3
isub	1.7	3.0	2.4	istore_2	1.4	3.1	2.2
ireturn	1.4	3.0	2.2	istore_1	2.1	2.0	2.0
iload_3	2.1	2.3	2.2	pop	2.3	1.8	2.0
iload	1.7	2.5	2.1	return	2.1	1.0	1.6
iaload	0.5	2.3	1.4	iload_2	1.1	1.8	1.5

Table 6: Total (API and non-API) dynamic bytecode execution frequencies for the Eiffel programs studied when compiled using SmallEiffel all_check mode.

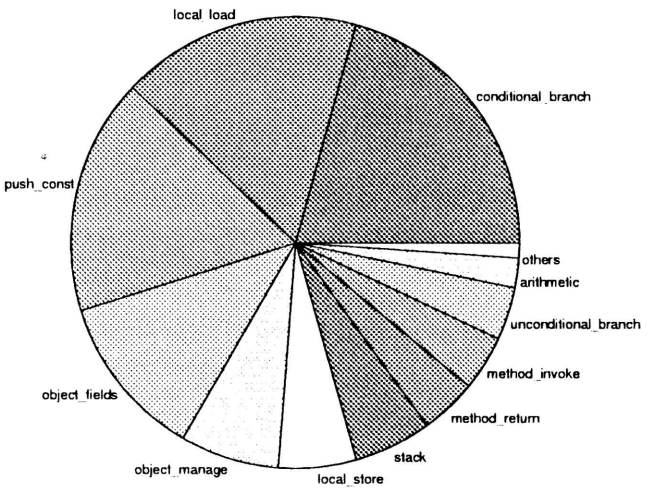

Figure 2: Total (API and Non-API) dynamic bytecode execution frequencies by category by the Eiffel programs studied when compiled using SmallEiffel all_check mode.

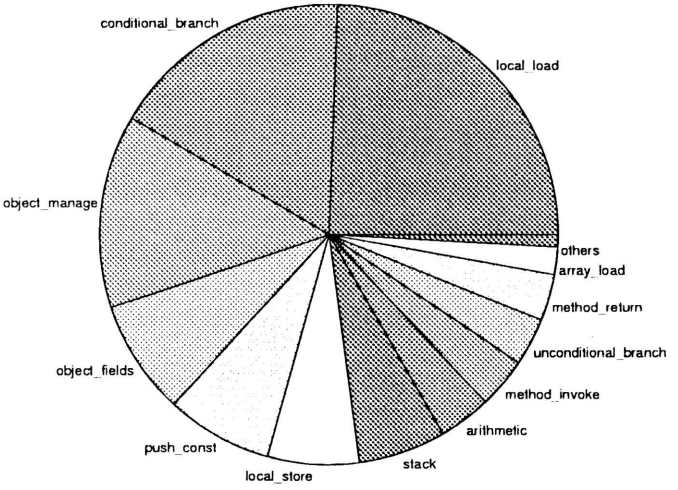

Figure 3: Total (API and Non-API) dynamic bytecode execution frequencies by category by the Eiffel programs studied when compiled using SmallEiffel no_check mode.

5 Stack Frame Usage

Each Java Virtual Machine thread has a private Java stack, created at the same time as the thread. A Java stack stores Java Virtual Machine frames. The Java stack is similar to the stack of a conventional language such as C. It holds local variables and partial results, and plays a part in method invocation and return. Because the stack is never manipulated directly except to push and pop frames in the Java design, it may actually be implemented as a heap, and Java frames may be heap allocated and do not need to be contiguous [11].

A Java Virtual Machine frame is used to store data and partial results, as well as to perform dynamic linking, to return values for methods, and to dispatch exceptions. A new frame is created each time a Java method is invoked. A frame is destroyed when its method completes, whether that completion is normal or abnormal (by throwing an exception). Each frame has its own set of local variables and its own operand stack. The memory space for these structures can be allocated simultaneously, since the sizes of the local variable area and operand stack are known at compile time and the size of the frame data structure depends only upon the implementation of the Java Virtual Machine [11].

On each Java method invocation, the Java Virtual Machine allocates a Java frame, which contains an array of words known as its local variables. Local variables are addressed as word offsets from the base of that array. Local variables are always one word wide. Two local variables are reserved for each long or double value. These two local variables are addressed by the index of the first of the variables [11]. The local variables holds three different categories of data:

- position zero holds the *this* pointer if executing an instance method
- the parameters, if any, used by the method.
- the temporary variables, if any, declared by the method.

The Java stack frame also contains an operand stack. Most Java Virtual Machine instructions take values from the operand stack of the current frame, operate on them, and return results to that same operand stack. The operand stack is also used to pass parameters to methods and receive method results. Subcomputations may be nested on the operand stack, resulting in values that can be used by the encompassing computation [11]. Unlike the stack used in a C program, the size of the Java operand stack is known at compile time.

This Section studies the way some real object oriented programs, written in Eiffel and Java used the stack frame of the Java Virtual Machine during execution. Dynamic measurements of local variable, parameter temporary variable and operand stack sizes were made for every method call during the execution of both program and API methods for the programs studied.

5.1 Dynamic Stack Frame Usage by Java Programs

Table 7 shows dynamic percentages of different local variable array, operand stack, temporary variable and parameter sizes for Java methods, including the *this* pointer for instance methods for the programs studied. Calls to native methods were not included when calculating these percentages. One surprising result is the high level of method calls with a local variable array size of zero made by both JavaCC and JJtree. The explanation for this seems to be that some of the methods used by these programs may not have been coded in true object oriented Java style, possibly for reasons of efficiency. The high number of method calls with local array size of one and two may be instance methods that get an object field and set an object field respectively, which can be typical of object oriented programs. 60% of method calls for the Java programs studied needed a local variable array size of two or less.

The high percentage of Java methods with zero operand stack size gives concern, as *nop* is the only bytecode that does not need an operand stack, but this can be explained by two factors. The Java/lang/Object/<init> method is ultimately called for each object that is created. This counts for 17,557 of the method calls with no operand stack during JJTree execution. There are an additional 1,099 calls to methods in the JJTree code that have no bodies, and 125 calls to Java API methods that have no bodies. One explanation for this might be an object oriented design with consistency of interface to allow for inheritance. 89% of methods executed needed an operand stack size of five of less. 75% of the methods executed do not have any temporary variables, which means local array size is largely needed for parameters.

5.2 Dynamic Stack Frame Usage by Eiffel Programs

Dynamic percentages of local variable array, operand stack, temporary variable and parameter sizes for Eiffel programs, including the this pointer for instance methods when compiled using SmallEiffel no_check and all_check modes are shown in Table 8.

The figures for the local variable array sizes are broadly the same for both modes, with a noticeable drop-off in numbers of methods with array size over 3. The differences between these figures and those for the Java programs in Table 5 can be sourced to the high parameter counts for the *atomic* and *fire works* Java programs, both of which make extensive use of parameter-heavy API routines.

The figures for the operand stack sizes show that considerable changes take place as a result of turning on all assertion-checking using the all_check mode. While the figures for no_check mode are broadly comparable to those for Java programs, the number of methods with operand stacks of size 6 and over appear unique to Eiffel programs compiled under all_check mode. These larger operand

	atom	fire	jas	jjt	jcc	f_i
Local variable array size						
0	0.3	0.4	0.2	11.8	10.4	4.6
1	24.9	18.8	42.9	30.5	27.2	28.9
2	23.4	28.4	24.6	32.1	30.4	27.8
3	1.4	0.7	9.6	5.4	8.4	5.1
4	1.0	21.3	6.0	9.7	8.0	9.2
5	46.3	26.1	7.8	6.2	9.2	19.1
6	1.3	3.0	7.2	2.2	1.4	3.0
7	0.1	0.0	0.6	0.4	1.6	0.5
8	0.1	0.0	0.9	1.5	3.0	1.1
9	0.1	0.0	0.0	0.0	0.1	0.0
> 9	1.0	1.3	0.0	0.3	0.3	0.6
Operand stack size						
0	1.6	1.7	6.0	7.6	5.9	4.6
1	2.5	3.7	19.2	12.1	11.5	9.8
2	25.0	28.8	19.4	26.1	24.4	24.7
3	3.6	5.9	34.2	12.7	13.7	14.0
4	19.5	2.8	9.2	20.7	20.3	14.5
5	10.9	55.6	9.9	14.0	16.5	21.4
6	0.6	0.6	1.2	4.2	3.5	2.0
7	0.1	0.5	0.0	2.0	3.4	1.2
8	35.2	0.0	0.9	0.2	0.3	7.3
9	0.4	0.0	0.0	0.4	0.3	0.2
> 9	0.6	0.4	0.0	0.0	0.1	0.2
Temporary variable size						
0	92.7	62.2	75.1	77.4	70.3	75.5
1	2.5	6.1	7.1	9.4	14.2	7.9
2	0.6	20.4	6.1	8.2	8.4	8.7
3	0.7	1.5	3.3	1.0	0.8	1.5
4	2.0	6.2	7.5	3.4	4.4	4.7
5	0.9	2.7	0.9	0.0	0.2	0.9
6	0.1	0.1	0.0	0.0	1.1	0.3
7	0.3	0.0	0.0	0.4	0.3	0.2
8	0.0	0.6	0.0	0.0	0.0	0.1
9	0.0	0.0	0.0	0.0	0.0	0.0
> 9	0.2	0.2	0.0	0.3	0.2	0.2
Parameter size						
0	0.8	1.3	0.2	16.8	14.9	6.8
1	27.9	32.9	47.6	38.6	36.3	36.7
2	23.7	44.3	34.8	28.2	30.1	32.2
3	1.4	0.4	12.8	1.4	0.8	3.4
4	0.4	0.7	3.2	11.0	14.4	5.9
5	44.5	19.6	1.4	3.9	3.0	14.5
6	0.4	0.1	0.0	0.0	0.1	0.1
7	0.3	0.0	0.0	0.0	0.1	0.1
8	0.1	0.0	0.0	0.1	0.2	0.1
9	0.1	0.0	0.0	0.0	0.0	0.0
> 9	0.5	0.6	0.0	0.0	0.1	0.2

Table 7: Dynamic percentages of different local variable array, operand stack, temporary variable and parameter sizes for Java methods, including the *this* pointer for instance methods.

no_check mode				all_check mode			
size	geyacc	gelex	f_i	size	geyacc	gelex	f_i
Local variable array size							
0	0.0	0.0	0.0	0	0.0	0.0	0.0
1	13.2	4.5	8.8	1	9.8	1.4	5.6
2	27.0	12.8	19.9	2	40.3	30.5	35.4
3	49.9	82.1	66.0	3	36.4	66.6	51.5
4	5.1	0.3	2.7	4	9.2	1.2	5.2
5	3.6	0.3	1.9	5	2.0	0.1	1.1
6	0.7	0.0	0.3	6	0.7	0.0	0.3
7	0.1	0.0	0.1	7	0.2	0.0	0.1
8	0.0	0.0	0.0	8	1.3	0.0	0.7
9	0.0	0.0	0.0	9	0.0	0.0	0.0
> 9	0.2	0.0	0.1	> 9	0.1	0.0	0.1
Operand stack size							
0	3.8	0.3	2.0	0	1.7	0.1	0.9
1	10.9	4.3	7.6	1	9.5	2.3	5.9
2	18.1	6.9	12.5	2	8.3	23.8	16.1
3	37.7	78.4	58.1	3	0.4	0.1	0.2
4	1.2	0.1	0.7	4	0.1	0.0	0.1
5	3.3	3.3	3.3	5	12.9	24.3	18.6
6	3.9	2.1	3.0	6	6.0	1.7	3.9
7	9.2	0.5	4.8	7	5.4	1.2	3.3
8	4.5	3.8	4.2	8	17.1	18.9	18.0
9	2.6	0.1	1.4	9	10.0	6.1	8.1
> 9	4.7	0.1	2.4	> 9	28.6	21.5	25.1
Temporary variable size							
0	32.1	11.5	21.8	0	14.0	3.2	8.6
1	61.1	88.2	74.7	1	62.0	73.3	67.7
2	4.2	0.2	2.2	2	14.7	23.3	19.0
3	1.5	0.1	0.8	3	7.0	0.1	3.5
4	0.8	0.1	0.5	4	0.5	0.1	0.3
5	0.2	0.0	0.1	5	1.7	0.1	0.9
6	0.0	0.0	0.0	6	0.0	0.0	0.0
7	0.0	0.0	0.0	7	0.0	0.0	0.0
8	0.0	0.0	0.0	8	0.1	0.0	0.1
9	0.1	0.0	0.1	9	0.0	0.0	0.0
> 9	0.0	0.0	0.0	> 9	0.0	0.0	0.0
Parameter size							
0	0.3	0.0	0.1	0	0.0	0.0	0.0
1	31.6	14.4	23.0	1	66.5	53.5	60.0
2	56.7	81.4	69.1	2	27.9	45.3	36.6
3	8.7	3.9	6.3	3	5.0	1.2	3.1
4	1.3	0.1	0.7	4	0.4	0.0	0.2
5	1.3	0.1	0.7	5	0.1	0.0	0.1
6	0.0	0.0	0.0	6	0.0	0.0	0.0
7	0.0	0.0	0.0	7	0.0	0.0	0.0
8	0.0	0.0	0.0	8	0.0	0.0	0.0
9	0.0	0.0	0.0	9	0.0	0.0	0.0
> 9	0.0	0.0	0.0	> 9	0.0	0.0	0.0

Table 8: Dynamic percentages of local variable array, operand stack, temporary variable and parameter sizes for Eiffel programs when compiled using SmallEiffel no_check and all_check modes.

stacks can be explained by the need to evaluate invariants, which are implemented as boolean-valued expressions, and whose complexity would typically be greater than those boolean-valued expressions normally used in conditional and iteration statements.

The proportion of methods with one temporary variable is considerably higher for the Eiffel programs studied than for any of the Java programs, irrespective of whether or not assertion checking is enabled. This can be explained by the mechanism used in Eiffel to return a value from a method, where the return value is always assigned into a special variable called `Result`. The SmallEiffel compiler implements this naively, treating it as though it were an ordinary local variable during the method's execution, and then pushing it onto the stack before the return instruction. Thus even simple accessor methods, which normally contain just a simple return statement, are translated to bytecode methods that have a temporary variable size of one.

As can be seen from the last section of Table 8, few of the Eiffel methods executed had no parameters. As mentioned earlier, this contrast with the figures for Java programs is more likely a result of the specific programs studied, rather than the language or compilation mode used. The effect of all_check mode on Eiffel programs can be seen in a reversal of the proportions of methods with one and two parameters, going from roughly a 1:3 ratio to almost a 2:1 ratio. The most likely explanation for this is that the class invariants are implemented by generating an extra method for each class called `invariant()`, which, as an instance method with no parameters, would have been recorded as a method with parameter size 1 in Table 8.

6 Conclusions

The results obtained from studying the test suite must be analysed with caution since they may have been influenced by the particular programs analysed. The JVM is primarily designed for Java, and it is difficult to translate a language like Eiffel which includes genericity as well as a complete assertion mechanism.

For the set of programs studied, most of the bytecodes are used at least once during execution. However a small subset of the bytecodes is executed with very high frequency. The data in this paper question the stack based design for the bytecode intermediate representation of Java programs. Because a stack based architecture is not random access, there is a lot of moving data between the stack and local variables which does not do any useful work but is merely a way of telling those instructions that do the work of the program what operands to use. Previous research [9] has shown that the categories local_load, push_const and local_store account for 44.9 percent of instructions executed for the Java programs analysed. This research shows that for the Eiffel programs studied, these three categories account for 39.9 percent of instructions when the no_check compiler mode is chosen, implying that this property is a function of the instruction set design, not the programming language used. When the all_check compilation mode is used and assertions are checked, the conditional branch instructions become the most frequently executed category. In spite of this local_load, push_const and local_store still account for 41.1% of instructions executed, a remarkably consistent figure.

The moving of values between local variable arrays and the operand stack uses a lot of execution time when bytecodes are interpreted. However for frequently used parts of a program, a dynamic compiler will incorporate a lot of these pushes and pops into a single native instruction. JIT compiling to convert this inefficient representation to native code has been proposed as a solution but to achieve a performance competitive with the code produced by a native compiler, it would be necessary to apply the same code-generation techniques which seems too expensive for JIT compilation.

The dynamic percentages of instructions and non-native method calls in the API and the program classes was studied to compare the way different applets and applications use the bytecodes and it was found that 74.1% of bytecodes executed and 67.1% of Java methods are in the API. This suggests a way to improve the speed of Java programs would be to compile the API methods to native instructions and save these on disk in a standard format, cutting the time spent interpreting programs. The Eiffel programs analysed however only spend a small percentage of there execution time in API methods, so this optimization would have relatively little effect on the speed of execution of Eiffel programs.

Stack frame usage was similar for both object-oriented languages. 66.4% of the methods in the Java programs studied used a local variable array size of 3 or less. With the no_check compiler option 84.7% of the Eiffel program's methods used a local variable array size of 3 or less, rising to 92.5% when all_check was used. Interestingly when the Eiffel compiler starts to perform checks, 25% of stack frames created need an operand stack greater than 9 words in size. These larger operand stacks can be explained by the need to evaluate invariants, which are implemented as boolean-valued expressions, and whose complexity would typically be greater than those boolean-valued expressions normally used in conditional and iteration statements.

References

[1] Gobo eiffel project. *<http://www.gobosoft.com/>*.

[2] Java compiler compiler website. *<http://www.suntest.com/JavaCC>*.

[3] Anonymous. The java hotspot performance engine architecture: A white paper about sun's second generation performance technology. <http://java.sun.com/products/hotspot/>.

[4] Denis Antonioli and Markus Pilz. Analysis of the java class file format. *Dept. of Computer Science, University of Zurich, Technical Report 98.4.*

[5] Eric Armstrong. Hotspot: A new breed of virtual machine. *Java World*, March 1998.

[6] Kenneth Bowles. Ucsd pascal. *Byte*, (46):170–173, May 1978.

[7] Lars Raeder Clausen. A java bytecode optimizer using side-effect analysis. *ACM 1997 Workshop on Java for Science and Engineering Computation*, June 21 1997.

[8] Colnet Dominique Collin Suzanne and Zendra Olivier. Type inference for late binding: The small eiffel compiler. *Joint Modular languages Conference JMLC '97. Springer Verlag Lecture Notes in Computer Science*, 1204:67–81, 1997.

[9] Waldron J. Dynamic bytecode usage by object oriented java programs. *Technology of Object-Oriented Languages and Systems 29th International Conference and Exhibition, Nancy, France*, June 1999.

[10] Gosling James. Oak intermediate bytecodes. *ACM SIGPLAN Workshop on Intermediate Representations (IR '95)*, pages 111–118.

[11] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison Wesley, 1996.

[12] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1997.

[13] John Meyer and Troy Downing. *The Java Virtual Machine.* O'Reilly, 1997.

[14] J Waldron. Dynamic bytecode data. *<http://www.compapp.dcu.ie/~jwaldron>.*

[15] Tim J Wilkinson. Kaffe, a virtual machine to run java code. *<www.kaffe.org>.*

[16] Colnet D Zendra O and Collin S. Efficient dynamic dispatch without virtual function tables. the smalleiffel compiler. *12th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'97)*, 32(10):125–141, October 1997.