Dynamic Profiling & Comparison of Sun Microsystems' JDK1.3.1 versus the Kaffe VM APIs

Author: Anthony Sartini
Computer Engineering, Trinity College, Dublin 2, Ireland
Supervisor: John Waldron
Department of Computer Science, Trinity College, Dublin 2, Ireland

1. Introduction

This research, which is part of my final year project, sets out to perform a dynamic method execution analysis to investigate the implementation of Sun Microsystems' JDK1.3.1 Virtual Machine and its APIs. The results obtained from this analysis are to be compared with those previously measured in [1] & [2] on Kaffe, in order to determine if useful results can be found, e.g. which JVM is more efficient, how do the APIs differ, etc.? Kaffe [3] is an independent implementation of the Java Virtual Machine which was written from scratch. It also comes equipped with its own class libraries; it should be noted that these libraries are not 100% compatible with Sun's JDK. It should also be noted that Sun's JDK1.3.1 was run using the Windows Operating System and Kaffe was run using the Linux Operating System. In order to test this technique, the Java Grande Forum Benchmark [4] and the SPEC JVM98 Benchmark [5] suites were used.

The remainder of this paper is organised as follows. Section 2 presents a method-level view of the dynamic profiles. Section 3 concludes the paper, and Section 4 presents a list of references used in this paper.

2. Dynamic Method Execution Frequencies

This section presents the dynamic profiles of the Java Grande Forum and SPEC JVM98 benchmarks based on methods, since these provide both a logical source of modularity at source-code level, as well as a possible unit of granularity for hotspot analysis [6]. It should be noted that since this analysis is carried out at the platform independent level, the behaviour inside native methods is not studied. Also, since the benchmark suites are written in Java, it's possible to conclude that all native methods are in the APIs.

Tables 1 and 2 show the most frequently executed API methods for the Grande programs, while Tables 3 and 4 show measurements of the total number of dynamic method calls, including native method calls, made during the execution of the Grande benchmark suite when running on both Sun and Kaffe. Tables 5 and 6 show the most frequently executed API methods for the SPEC programs, while Tables 7 and 8 show measurements of the total number of dynamic method calls, including native method calls, made during the execution of the SPEC JVM98 benchmark suite on both Sun and Kaffe.

Sun's JDK1.3.1

Method Name	Frequency
Euler	
java.lang.Math.abs	86.0
java.lang.StrictMath.log*	1.1
java.lang.Math.log	1.1
java.lang.StrictMath.pow*	1.1
java.lang.Math.pow*	1.1
Moldyn	
java.lang.StrictMath.log*	19.8
java.lang.Math.log*	19.8
java.lang.String.charAt	11.1
java.lang.StringBuffer.append	5.3
java.lang.String.indexOf*	3.6
Montecarlo	
java.util.Random.next	25.6
java.lang.StrictMath.log*	15.1
java.lang.Math.log*	15.1
java.util.Random.nextDouble	12.8
java.util.Random.nextGaussian	10.1
Raytracer	
java.lang.Math.abs	87.4
java.lang.StrictMath.pow*	5.7
java.lang.Math.pow*	5.7
java.lang.String.charAt	0.2
java.lang.System.arraycopy*	0.1
Search	
java.lang.String.charAt	19.7
java.lang.StringBuffer.append	9.8
java.lang.String.indexOf*	6.9
java.lang.System.arraycopy*	5.8
java.lang.String.<init>	3.8

Kaffe

Method Name	Frequency
Euler	
java/lang/Math.abs	42.3
java/lang/Object.<init>	33.8
java/lang/Math.sqrt*	19.8
java/lang/Math.pow*	0.5
java/lang/Math.log	0.5
Moldyn	
java/lang/Math.sqrt*	84.5
java/lang/Math.log*	4.3
java/lang/Object.<init>	2.5
java/lang/String.indexOf	2.4
java/lang/System.arraycopy*	0.6
Montecarlo	
java/util/Random.next	31.9
java/lang/Math.log*	18.8
java/util/Random.nextDouble	16.0
java/util/Random.nextGaussian	12.5
java/lang/Math.exp*	12.5
Raytracer	
java/lang/Math.sqrt*	52.1
java/lang/Object.<init>	41.4
java/lang/Math.abs	6.0
java/lang/Math.pow*	0.4
java/lang/String.indexOf	0.0
Search	
java/lang/String.indexOf	40.3
java/lang/Object.<init>()	6.4
java/lang/StringBuffer.append	5.8
java/lang/String.<init>	3.2
java/util/HashMap.bucket	1.8

Tables 1 & 2: Dynamic method execution frequencies for the most heavily used API methods for the Grande applications, including native methods. Native methods are indicated by *.

Tables 1 and 2 above illustrate the most heavily invoked API methods for each of the Grande programs. It can be seen from the tables that some of the methods that are invoked for the Grande programs on Sun, differ from those invoked on Kaffe. This is to be expected since the APIs have been implemented differently for both Sun and Kaffe. The figures in the tables indicate that Kaffe concentrates the majority of its API method calls to the top five methods invoked, whereas Sun appears to distribute its method calls across all of the API methods invoked.

Sun's JDK1.3.1

Program	Methods	API %	API nat %
Eul	2.35e + 07	40.4	2.6
Mol	4.40e + 05	3.5	1.7
Mon	1.00e + 08	99.1	50.0
Ray	4.44e + 08	0.2	0.0
Sea	7.12e + 07	0.0	0.0
Average	1.28e + 08	28.6	10.9

Kaffe

Program	Methods	API %	API nat %
Eul	3.34e + 07	58.0	12.6
Mol	5.49e + 05	22.7	19.9
Mon	8.07e + 07	98.7	37.4
Ray	4.58e + 08	3.1	1.6
Sea	7.12e + 07	0.0	0.0
Average	1.29e + 08	36.5	14.3

Tables 3 & 4: Measurements of the total number of method calls, including native calls, by Grande applications. Also shown is the percentage of the total which are in the API, and percentage of total which are native methods.

It can be seen from Tables 3 and 4 that, for virtually every Grande program, Sun invokes fewer methods overall and a smaller percentage of API methods than Kaffe. Due to the fact that calls to non-API methods (i.e. method calls local to each of the Grande programs) on both Sun and Kaffe were identical, it should be clear that the reduction in overall method invocations on Sun's VM is due to a smaller number of API method invocations being made by Sun. Since the Search program invoked a negligible amount of API methods, the method call frequencies are identical.

The reasons as to why Sun invokes fewer methods overall for each of the benchmark programs are as follows:

- The method *java.lang.Math.sqrt* is invoked heavily on Kaffe but not on Sun; it is either being inlined by Sun's HotSpot™ VM or it's being repla ced by an efficient piece of native code
- The method *java.lang.Object.<init>* is also invoked heavily on Kaffe but not on Sun; perhaps Sun have realised that due to the fact that this method actually does nothing, time is wasted fetching this method and thus, has been removed from Sun's VM

Although the total method invocations, on average, for both Sun and Kaffe are very close, Sun uses approximately 8% fewer API methods than Kaffe. This is a reasonable hypothesis that suggests Sun's VM is performing more efficiently than Kaffe.

Sun's JDK1.3.1

Method Name	Frequency
compress	
java.lang.String.charAt	14.0
java.lang.StringBuffer.append	4.8
java.lang.System.arraycopy*	4.6
java.lang.String.hashCode	4.1
java.lang.String.indexOf*	3.9
db	
java.util.Vector.elementAt	50.0
java.lang.String.compareTo	25.0
java.util.Vector$1.nextElement	9.1
java.util.Vector$1.hasMoreElements	7.4
java.util.Vector$1.<init>	3.2
mtrt	
java.io.BufferedInputStream.ensureOpen	31.4
java.io.BufferedInputStream.read	31.4
java.lang.String.charAt	9.7
java.lang.String.substring	3.4
java.lang.String.<init>	3.3
jack	
java.lang.Object.equals	6.1
java.lang.System.arraycopy*	4.6
sun.io.CharToByteSingleByte.getNative	3.9
java.lang.String.charAt	3.9
java.lang.String.hashCode	3.6
javac	
java.io.BufferedInputStream.ensureOpen	17.3
java.io.BufferedInputStream.read	17.3
java.util.Hashtable.get	5.3
java.lang.System.arraycopy*	4.1
java.lang.Object.hashCode*	3.6
mpegaudio	
java.lang.System.arraycopy*	92.6
java.lang.Math.min	4.8
java.io.FilterInputStream.available	1.2
java.lang.Math.max	0.9
java.lang.String.charAt	0.3
jess	
java.lang.String.equals	15.2
java.lang.System.arraycopy*	14.1
java.lang.String.hashCode	11.5
java.lang.Number.<init>	7.6
java.lang.Integer.equals	7.6

Kaffe

Method Name	Frequency
compress	
java/lang/String.indexOf	6.2
java/util/HashMap.bucket	5.1
java/util/HashMap$Entry.access$1	3.7
java/util/HashMap$Entry.access$0	3.3
java/lang/StringBuffer.append	2.4
db	
java/util/Vector.elementAt	36.9
java/lang/String.compareTo	18.4
java/lang/Math.min	18.4
java/util/Vector$1.nextElement	6.7
java/util/Vector$1.hasMoreElements	5.5
mtrt	
java/lang/Object.<init>	58.2
java/io/FilterInputStream.read	7.6
java/io/DataInputStream.read	7.1
java/lang/StringBuffer.append	2.0
java/lang/Float.isNaN	0.3
jack	
java/lang/Object.equals	9.0
java/util/HashMap.access$1	5.3
java/util/Vector.size	3.9
java/util/Vector.<init>	3.8
java/util/HashMap.find	3.3
javac	
java/io/BufferedInputStream.read	17.2
java/util/HashMap.find	5.8
java/lang/Object.equals	4.7
java/lang/Object.<init>	4.2
java/util/HashMap.get	3.4
mpegaudio	
java/lang/Math.min	4.5
java/io/FilterInputStream.available	1.1
java/lang/Math.max	0.9
java/lang/System.currentTimeMillis*	0.6
java/lang/String.hashCode	0.3
jess	
java/util/HashMap.find	16.0
java/lang/Object.<init>	12.1
java/lang/String.equals	11.6
java/util/HashMap.get	8.1
java/util/HashMap.bucket	8.0

Tables 5 & 6: Dynamic method execution frequencies for the most heavily used API methods for the SPEC JVM98 applications, including native methods. Native methods are indicated by *.

On inspection of Tables 5 and 6, on the previous page, it can be seen that the majority of the most frequently executed API methods for each program, appear to be different for both Sun and Kaffe. Again, this is no surprise as Kaffe's APIs are implemented independently of Sun's APIs. Since the calls to non-API methods (i.e. method calls local to each of the SPEC programs) on both Sun and Kaffe were identical, with the exceptions of 'mtrt' and 'db', it should be clear that the total API method calls will again be the governing factor in determining the overall method invocations for each program.

Perhaps the rationale behind the 'mtrt' and 'db' programs' non-API method frequencies differing between Sun and Kaffe is:

- 'mtrt' uses threads during its execution and these threads may be handled differently on Linux and Windows

- The methods that differ between Sun and Kaffe, for 'db', originate from the Harness and IO classes (responsible for file handling); perhaps the methods in these classes execute diversely on Linux and Windows.

Sun's JDK1.3.1

Program	Methods	API %	API nat %
cmprs	2.25e + 08	0.0	0.0
db	9.20e + 07	98.3	0.0
mtrt	2.29e + 07	1.0	0.0
jack	4.94e + 07	85.1	5.1
javac	9.57e + 07	50.7	4.5
mpeg	9.94e + 07	1.3	1.2
jess	9.61e + 07	19.2	2.8
Average	9.72e +07	36.5	1.9

Kaffe

Program	Methods	API %	API nat %
cmprs	2.26e + 08	0.0	0.0
db	1.24e + 08	98.7	0.1
mtrt	2.88e + 08	3.2	0.1
jack	1.16e + 08	92.3	4.2
javac	1.53e + 08	62.0	2.8
mpeg	1.10e + 08	1.3	1.1
jess	1.35e + 08	32.5	1.9
Average	1.65e + 08	41.4	1.5

Tables 7 & 8: Measurements of the total number of method calls, including native calls, by SPEC applications. Also shown is the percentage of the total which are in the API, and percentage of total which are native methods.

On comparison of Tables 7 and 8 above, the first noticeable difference between them is the total number of method invocations made during the execution of each of the SPEC programs. The total number of method invocations has considerably reduced for each program when executed on Sun. The major difference can be seen in 'jack'; Sun executes approximately 66.6 million fewer methods than Kaffe in this program. Again, since all non-API method invocation frequencies were identical for the programs, on both Sun and Kaffe, this reduction in total method calls is due to the number of API method invocations made by Sun. Again, the decrease in API percentages is partially due to the same reasons as mentioned above for the Grande suite.

Although Sun invokes a slightly higher number of native-API methods than Kaffe, on average it uses approximately 5% fewer API methods overall, which indicates that the former virtual machine reduces the number of methods invoked and hence reduces the time spent fetching methods. Again, this is a plausible postulation which leans towards Sun's VM as the more efficient virtual machine.

However, it should be noted that although Sun invokes fewer methods dynamically than Kaffe, for each of the two benchmark suites' programs, it is not entirely correct to conclude that Sun is more efficient than Kaffe just because it invokes fewer methods. Some of the methods invoked by Sun may execute more bytecodes than those invoked on Kaffe. In order to provide a more concrete foundation as to if and why Sun is more efficient than Kaffe, a dynamic bytecode analysis needs to be performed for the Grande and SPEC benchmark suites' programs.

3. Conclusion

This paper set out to perform a dynamic API analysis and comparison of two diverse implementations of the Java Virtual Machine, namely Sun's JDK1.3.1 and Kaffe. It has been shown that useful information can be found by performing a dynamic method profile of the two benchmark suites. Sun's VM executes fewer methods overall than Kaffe for the Grande and SPEC suites, which is due to fewer API method invocations being made by Sun. This is a key indication that Sun Microsystems have put a good deal of effort into optimising their VM by reducing the amount of API methods being fetched, and therefore, reducing the execution time of the programs. In conclusion, the results presented in this paper indicate Sun Microsystems's VM to be more efficient then the Kaffe VM.

My research, with regards Sun versus Kaffe, is still in progress and further analysis of the two VMs has included a *dynamic bytecode analysis*. The results from this analysis have shown that Sun executes fewer bytecodes dynamically than Kaffe for the benchmark suites' programs. This is a more valid reason as to why Sun is more efficient than Kaffe.

4. References

1. Daly C, Horgan J, Power J, Waldron J, "Platform Independent Dynamic Java Virtual Machine Analysis: the Java Grande Forum Benchmark Suite". In *Joint ACM Java Grande – ISCOPE 2001 Conference*, pages 106-115, Stanford, CA, USA.

2. Gregg D, Power J, Waldron J, "Benchmarking the Java Virtual Architecture In Java Microarchitectures", edited by Vijaykrishnan Narayanan and Mario L. Wolczko Kluwer Academic Publishers, Boston. April 2002 (ISBN 1-4020-7034-9), pages 1-19

3. T. J. Wilkinson, "KAFFE, A Virtual Machine to run Java Code", www.kaffe.org.

4. Java Grande Forum Benchmark Suite, http://www.epcc.ed.ac.uk/javagrande

5. SPEC JVM98 Benchmark Suite, http://ww.specbench.org/osg/jvm98

6. Sun Microsystems (2001), "The Java HotSpot™ Virtual Machine: Technical Whitepaper".