A Quantitative Analysis of Java Bytecode Sequences

Ben Stephenson
Wade Holst
Department of Computer Science, University of Western Ontario, London, Ontario, Canada

1 Introduction

A variety of studies have been performed which analyse the bytecodes executed by a Java Virtual Machine (JVM). The simplest of these studies perform a static analysis of the bytecodes in the classes that make up the program [1]. Other studies have examined the dynamic behaviour of the program, only considering those individual bytecodes that are actually executed [2, 3]. Dynamic studies have also been extended to determine which bytecode pairs are commonly executed [4]. This work builds on these previous studies by extending the dynamic analysis from bigrams (bytecode pairs) to multicodes[1] (variable length sequences) up to 20 bytecodes in length. The algorithms used to determine the multicodes are presented in addition to the most commonly occurring multicodes for a selection of benchmarks. Determining these multicodes is relevant to research into instruction set design. It is also directly applicable to interpreter optimization techniques such as super operators [5] and just-in-time compiler optimization techniques including bytecode idioms [6].

2 Identifying Bytecode Sequences

We define a multicode to be a series of bytecodes that are always executed in sequence by the JVM such that every bytecode resides within the same method. Furthermore, any bytecode that is the target of any branch is only permitted to start a multicode. These constraints are employed to ensure that the multicodes identified will be of use for optimization.

In addition to constraining the multicodes of interest, it is also desirable to ensure that each occurrence of a bytecode is only counted once when the multicodes are being identified. If the same occurrence of a bytecode is included in two or more multicodes, the list of most frequently occurring multicodes will become invalid after any optimization that can only been applied to each bytecode once. For example, both super operators and bytecode idioms are optimization techniques that only allow each occurrence of a bytecode to be optimized once.

The algorithm illustrated in Figure 1 was used to identify multicodes. It begins by making use of a modified version of the Kaffe virtual machine [7] that records an execution trace of the application as it runs. This trace consists of the name of each method invoked, a description of the branch structure of each method invoked in addition to the program counter and opcode for every bytecode executed. This information is recorded for methods in the classes that make up the application in addition to the Java class library.

Using the execution trace, it is possible to identify the basic blocks in the sequence of bytecodes executed by the application. By breaking the program into basic blocks, we ensure that the constraints on multicode outlined above are met. Once the blocks have been identified it is possible to determine all of the multicodes executed by the application by finding all sequences of bytecodes of lengths 2 through 20 within each block. A counter associated with each sequence is incremented every time the sequence is identified.

[1] We elect to use the term *multicode* rather than *multigram* to emphasize the fact that a sequence of Java bytecodes is being discussed.

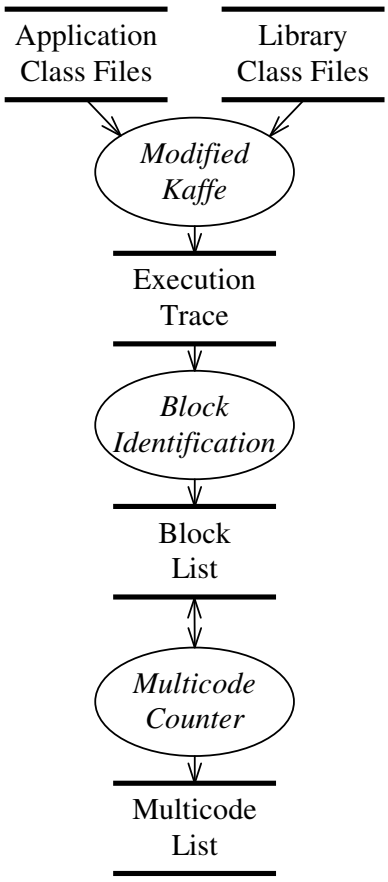

Figure 1: Sequence Identification Algorithm

For example, when the block of bytecodes `aload_0 getfield aload_0 putfield` is processed, three length 2 multicodes are identified (`aload_0 getfield`, `getfield aload_0` and `aload_0 putfield`), in addition to the length 3 multicodes (`aload_0 getfield aload_0` and `getfield aload_0 putfield`) and the multicode of length four.

Once all of the multicodes have been identified and counted, it is possible to determine which multicode is best. This could be determined by simply considering the number of occurrences of each multicode. However, this approach is biased towards shorter sequences because they are inherently more common than longer sequences. As a result, a score is determined for each multicode by multiplying the number of occurrences of the multicode by its length. This score is the total number of bytecodes that will be impacted by the use of the multicode.

The best multicode is the multicode with the highest score. It is recorded in the multicode list. All occurrences of the multicode are removed from the block list and replaced with a bytecode value that is unused in the Java Virtual Machine Specification [8]. This replacement is performed to reflect the fact that the bytecodes in the multicode have already been counted.

Now the process of identifying the best multicode is repeated using the modified block list. Each time a new multicode is identified, it is replaced with a new unused value and added to the multicode list. Note that it is possible that a value used previously to represent a multicode may appear as part of another multicode identified later. However, the multicode identified previously only counts as a single bytecode when the length of the new multicode is determined. As a result, the bytecodes that make up the multicode identified previously are not double counted with respect to determining which multicodes are best.

3 Results

Thirteen different benchmarks were used during testing. They include six benchmarks from the SPEC JVM98 Suite [9], the Linpack floating point benchmark [10], a 'Hello World' application and five benchmarks from The Great Computer Language Shootout [11]. A brief description of each benchmark is provided in Table 1.

Table 2 shows some additional statistics about each of the benchmarks. It begins by identifying the total number of bytecodes executed by each benchmark. Note that this count includes both the bytecodes executed by the application's classes and the Java libraries. As a result, these counts will vary depending on the library implementation used. Our testing was performed using a modified version of Kaffe 1.0.7 on a 650 MHz dual processor Pentium III with 512 megabytes of RAM running Linux version 2.4.20-20.9smp. The SPEC benchmarks were not recompiled. The remaining benchmarks were compiled using Kaffe's implementation of javac. In all cases, the class libraries included with Kaffe 1.0.7 were employed.

The third column shows the number of distinct bytecodes executed by each application. It is interesting to note that none of the applications execute all 201 bytecodes defined in the Java Virtual Machine Specification [8]. HELLO executes 109 distinct bytecodes, or 54%

of the total number possible, while MPEG executes 162 distinct bytecodes, or 81% of the total number possible.

The final two columns show the total number of multicodes that could theoretically occur during the execution of the application and the number of multicodes that were actually present. The number of theoretically possible sequences was computed as the sum of the number of unique bytecodes executed by the benchmark raised to the powers 2 through 20. MPEG had the largest number of sequences present with 83186 while only 4902 sequences were present in HELLO. In all cases, the number of multicodes that can theoretically occur is many orders of magnitude largers the number of multicodes that were actually encountered.

Benchmark	**Abbr.**	**Description**	**Configuration**
_201_compress	COMP	A compression / decompression program.	Input Size 1
_202_jess	JESS	An expert shell system.	Input Size 1
_209_db	DB	A in-memory database.	Input Size 1
_213_javac	JAVAC	A Java compiler.	Input Size 1
_222_mpegaudio	MPEG	MPEG-3 audio compressor.	Input Size 1
_228_jack	JACK	A Java parser generator.	Input Size 1
Linpack	LPACK	A solver for a system of linear equations	500x500 matrix
SO Count	WC	Counts lines, words and characters in provided text.	Default input file
SO Heapsort	HEAP	In-place heapsort of double precision numbers.	1000 array elements
SO Matrix	MAT	Performs matrix multiplication	30x30 matrices, 20 iterations
SO Spellcheck	SPELL	Spell checks words in a sample of text.	Default input file and dictionary, 20 iterations
SO Word Freq.	WORD	Number of times each word appears in a sample of text.	Default input file
Hello World	HELLO	Displays ‘Hello World!’	

Table 1: Benchmark Descriptions

Benchmark	Bytecodes Executed	Unique Bytecodes Executed	Theoretically Possible Multicodes	Number of Unique Multicodes
COMP	955194285	147	2.20E+43	15792
DB	2743724	140	8.40E+42	10639
HEAP	787687	137	5.50E+42	9652
HELLO	385727	109	5.70E+40	4902
JACK	69040616	136	4.70E+42	12970
JAVAC	11777531	152	4.40E+43	18157
JESS	12163198	139	7.30E+42	15838
LPACK	8336215	142	1.10E+43	10233
MATR	10384910	121	4.60E+41	7211
MPEG	120020905	162	1.60E+44	83186
SPELL	9736875	120	3.90E+41	6795
WC	763107	120	3.90E+41	7012
WORD	45775260	119	3.30E+41	6991

Table 2: Benchmark Properties

Seq.	Length	Multicodes for COMP
1	11	aload_0 getfield astore aload getfield iload_2 iaload istore iload iload_1 if_icmpne
2	2	aload_0 getfield
3	6	dup getfield dup_x1 iconst_1 iadd putfield
4	19	aload_0 dup getfield iconst_1 iadd putfield iload_3 S2 ishl iload iadd istore_1 iload_3 iload ishl iload ixor istore_2 S1
5	15	S2 S2 astore aload dup getfield iconst_1 isub putfield aload getfield aload getfield baload invokevirtual
6	18	S2 astore S2 astore aload getfield iload_2 baload istore aload getfield aload S3 iload bastore S2 iload_2 invokevirtual
7	13	aload_0 dup getfield iconst_1 isub putfield S2 aload_0 S3 baload sipush iand ireturn
8	8	S2 iload_1 saload bipush ishl bipush iushr ireturn
9	6	iload_2 iload isub dup istore_2 ifge
10	6	S2 aload_0 S3 iload_1 bastore return
Seq.	Length	Multicodes for HEAP
1	13	aload_1 iload aload_1 iload_3 daload dastore iload_3 iload_3 dup istore iadd istore_3 goto
2	10	aload_1 iload_3 daload aload_1 iload_3 iconst_1 iadd daload dcmpg ifge
3	4	aload_0 getfield aload_0 getfield
4	6	dload aload_1 iload_3 daload dcmpg ifge
5	9	aload_0 dup getfield iconst_1 iadd putfield S3 arraylength if_icmpge
6	4	iload aload_0 getfield if_icmplt
7	6	S3 iload iadd caload iload_3 if_icmpne
8	3	iload_3 iload if_icmple
9	5	iload_1 i2c istore_3 iload_2 ifge
10	5	S3 aload_0 getfield caload invokevirtual
Seq.	Length	Multicodes for LPACK
1	16	aload iload iload iadd dup2 daload dload_2 aload iload iload iadd daload dmul dadd dastore iinc
2	3	iload iload_1 if_icmplt
3	20	sipush iload imul ldc irem istore aload_1 iload aaload iload iload i2d ldc2_w dsub ldc2_w ddiv dastore aload_1 iload aaload
4	12	aload iload dup2 daload aload_1 iload aaload iload daload dadd dastore iinc
5	16	aload_2 iload dup2 daload aload iload daload aload iload aaload iload daload dmul dadd dastore iinc
6	6	S3 iload daload dload dcmpl ifle
7	3	iload iload_3 if_icmplt
8	12	aload_0 iload_3 iload isub dload aload iload iconst_1 aload iload iconst_1 invokevirtual
9	11	aload_1 iload aaload astore aload iload daload dstore iload iload if_icmpeq
10	10	aload iload aload iload daload dastore aload iload dload dastore
Seq.	Length	Multicodes for WORD
1	2	aload_0 getfield
2	8	S1 aload_0 dup getfield dup_x1 iconst_1 iadd putfield
3	6	S1 astore_1 aload_1 monitorenter aload_0 invokespecial
4	3	aload_0 iload_1 invokespecial
5	4	S1 iload_1 aaload areturn
6	7	aload iload_3 caload aload iload caload if_icmpeq
7	2	aload_0 invokespecial
8	4	astore_2 aload_1 monitorexit ret
9	9	bipush iload_1 imul S1 iload_3 caload iadd istore_1 iinc
10	3	iload_1 sipush if_icmple

Table 3: The 10 Best Bytecode Sequences for COMP, HEAP, LPACK and WORD

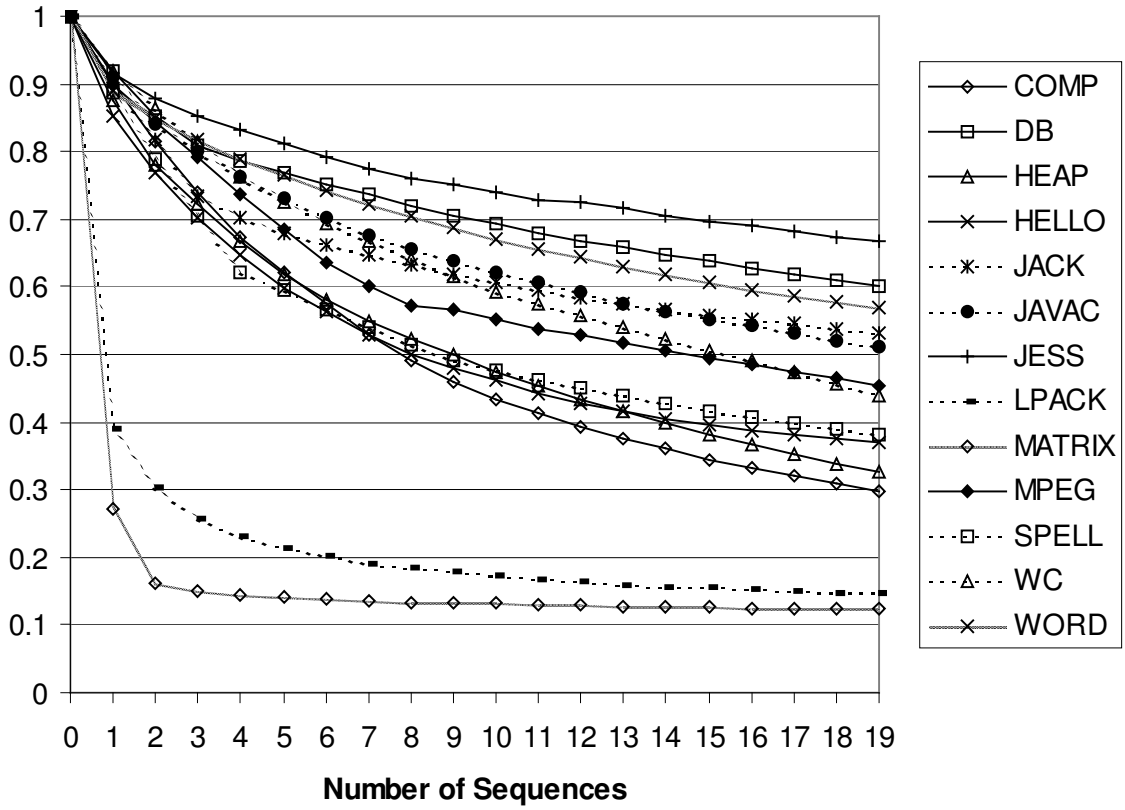

Figure 2: The Proportion of Bytecodes Remaining for Consideration after each Sequence is Identified.

In addition to presenting summary statistics for each of the benchmarks, we also provide lists of the best multicodes for a subset of the benchmarks as shown in Table 3. Results for the remaining benchmarks could not be included due to space constraints but are available from the authors. It is interesting to note that many longer multicodes are present. Only HELLO and WORD contain no multicodes of length 10 or more among their 10 best sequences. As discussed earlier, it is possible that sequences that have been previously identified may be part of a subsequent multicode. The letter 'S', followed by a number indicating the previously identified sequence, marks the use of these sequences.

It is interesting to note that many of the benchmarks share a similar bytecode sequence within some of their multicodes. Of those benchmarks listed in Table 3, all except LPACK make use of a multicode containing `aload_0 dup getfield [dup_x1] iconst_1 iadd [dup_x1] putfield`. This bytecode sequence is generated when a field is incremented. An extra `dup_x1` instruction may be included either immediately after the `getfield` or before the `putfield` to save the original or updated field value for a subsequent computation. When the 20 best multicodes for each benchmark are considered, only MPEG does not make use of this pattern. As a result, optimizers should give this sequence special consideration. Furthermore, instruction set designers should consider introducing a single bytecode for adjusting the value of a field if the Java bytecode language is revised.

In addition to identifying the best multicodes for each benchmark, we also examine the number of occurrences of each multicode as a proportion of the total number of bytecodes executed by the application. This is illustrated in Figure 2, which shows the proportion of the bytecodes remaining for consideration after each multicode is identified. It is interesting to note that this graph clearly illustrates that those sequences identified first make use of the most bytecodes. While this is not surprising, it is an important result because it indicates that a large proportion of the benefit of multicodes can be achieved using a relatively small num-

ber of sequences. This result for multicodes of lengths 2 through 20 is consistent with the result found for bigrams in [4].

4 Conclusion

An algorithm was presented for identifying the best multicodes for an application. This algorithm ensures that the multicodes identified are contained within a basic block and each occurrence of a bytecode is counted only once. The best multicodes of lengths 2 through 20 have been identified for a selection of benchmarks. It was found that only a very small proportion of those multicodes that are theoretically possible actually occur. Furthermore, it was also found that the top few multicodes make the greatest contribution.

Twelve of the thirteen benchmarks under consideration contained a variation of a sequence of bytecodes used to increment a field within one or more of their top 20 multicodes. This suggests that optimizers should give special consideration to this frequently occurring sequence. Instruction set designers should also consider including a field adjustment bytecode if the Java bytecode is extended.

We believe that additional work in this area is warranted. Superoperators and bytecode idioms should be implemented using the sequences identified here to demonstrate the sequences suitability. The scoring system used to determine which sequences are best could also be revised to determine the best multicodes based on their optimization potential in addition to their length and frequency.

5 Acknowledgments

The authors would like to thank NSERC for their generous support of this project.

6 References

[1] D.N. Antonioli and M. Pilz. *Analysis of the Java Class File Format*. Technical Report ifi-98.04, Department of Computer Science, University of Zurich, 1998.

[2] C. Daly, J. Horgan, J. Power, and J. Waldron. *Platform Independent Dynamic Java Virtual Machine Analysis: the Java Grande Forum Benchmark Suite*. Joint ACM Java Grande ISCOPE 2001 Conference. Stanford University, USA, 2001.

[3] C. Herder and J.J. Dujmovic. *Frequency Analysis and Timing of Java Bytecodes*. Technical Report SFSU-CS-TR-00.02, Computer Science Department, San Francisco State University, 2000.

[4] D. O'Donoghue, A. Leddy, J. Power, and J. Waldron. *Bigram Analysis of Java Bytecode Sequences*. The Inaugural Conference on the Principles and Practice of Programming in Java. Dublin, Ireland, 2002.

[5] T.A. Proebsting. *Optimizing an ANSI C Interpreter with Superoperators*. Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. San Francisco, California, 1995.

[6] T. Suganuma, et al., *Overview of the IBM Java Just-in-Time compiler.* IBM System Journal, 2000. **39**(1): p. 175-193.

[7] *Kaffe: A Clean Room Implementation of the Java Virtual Machine*. www.kaffe.org, 2004

[8] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*. Second Edition ed. The Java Series, ed. L. Friendly. 1999: Addison-Wesley.

[9] Standard Performance Evaluation Corporation, *SPEC Java Virtual Machine Benchmark Suite*. http://www.spec.org/jvm98,

[10] J. Dongarra, R. Wade, and P. McMahan, *Linpack Benchmark - Java Version*. http://www.netlib.org/benchmark/linpackjava/, 2004

[11] D. Bagley, *The Great Computer Language Shootout*. http://www.bagley.org/~doug/shootout/index2.shtml, 2004