An Analysis of Basic Blocks within SPECjvm98 Applications

Jonathan Lambert
Computer Science Dept.
National University of Ireland
Maynooth, Co. Kildare, Ireland
jonathan@cs.nuim.ie

James F. Power
Computer Science Dept.
National University of Ireland
Maynooth, Co. Kildare, Ireland
jpower@cs.nuim.ie

November 16, 2005

ABSTRACT

In this report we present the results of performing a quantitative analysis of basic blocks that are contained within SPECjvm98 applications. We first investigate the distribution of basic block sizes both statically and dynamically and then focus on frequently occurring basic blocks. We then present our findings on the workload differences between top ranked basic blocks and overall application. Finally we present the results from the investigation into the existence of a linear correlation between static and dynamic basic block frequencies.

Keywords

Java Virtual Machine, Basic block analysis, SPECjvm98 applications

1. INTRODUCTION

The Java Programming Language [11] has gained widespread popularity since it was first envisaged by its developers at Sun Microsystems in early 1995. This popularity was in part due to the Java Programming Language's associated intermediate format and underlying runtime environment, and it initially found favour and use as a secure and platfrom independent language representation that could be used by web browsers.

The Java Programming Language is composed of two environments, the compile time environment and the runtime environment. The Java compile time environment's ultimate function is to produce well formed Java class files that adhere to the Java class file specification as set out in [16]. The Java Runtime Environment (JRE), which includes the Java Virtual Machine (JVM) [16], is responsible for the loading and execution of Java class files.

The JVM is an abstract computing machine [16] and is similar to a real computer, in that it responds to a particular set of instructions in a well defined manner, and executes a sequence of prerecorded instructions that are contained within Java class files. Implementations of the JVM may interpret, just-in-time-compile or even execute these Java instructions natively. The Java class file format as specified in [16] defines 201 instructions, with three more special instructions used internally by the JVM.

An analysis of Java class file instructions can be performed both statically and dynamically, and the results of such analysis has been presented in the literature. The quantitative analysis of Java bytecode has contributed to JVM optimisations as well as contributing towards Java application modelling.

In this paper we abstract away from direct instruction level analysis and perform our analysis on basic blocks that are contained within Java bytecode.

This paper is structured as follows. In Section 2, we briefly describe the background of basic blocks and outline similar work that has been undertaken. Then in Section 3, we describe the benchmark applications and also the methodology. Next in Section 4, we present a small example on the calculation of basic blocks. In Section 5, we present the results on dynamic and static basic block sizes. In Section 6, we present our results on frequently executed basic blocks and their makeup. In Section 7, we consider if frequently occurring basic blocks are representative of the workload associated with an application. In Section 8, we relate static and dynamic basic block frequencies. Finally, in Section 9, we present our conclusions and future direction.

2. BACKGROUND AND RELATED WORK

The Java Programming Language is an object oriented programming language and thus allows a program to be viewed as a collection of individual units known as objects. Like many other forms of programming paradigms, Java allows the programmer to create a number of different types of statements within source code, some of which are: assignment statements, expression evaluation, looping, conditional branching and unconditional branching by the invocation of methods. The compilation of Java source code to its intermediate class file format transforms these high level statements to corresponding statements of bytecode instructions. Associated with conditional branching are sequences or blocks of instructions that are executed only if a condi-

Technical Report
NUIM-CS-TR-2005-15
Department of Computer Science
National University of Ireland
Maynooth, Co. Kidare, Ireland.

tion is evaluated to be true or false. Unconditional branching like the "goto" instruction or a method invocation such as an "invokevirtual", "invokestatic", "invokeinterface" or "invokespecial" also cause a change in program control and are associated also with sequences of instructions.

We define a basic block within a Java method's code to be a sequence of Java instructions, such that if the sequence's first instruction is executed by the JVM then all other instructions in the sequence is executed sequentially, if no runtime errors or exceptions occur. For a general definition of a basic block in an imperative programming language see [28].

Basic block analysis is not a new concept, Gregg et al. in [13] undertook an analysis of basic block sizes and frequencies that occur statically and dynamically within Forth programs. Maierhofer et al. in [17] present their empirical results on basic block sizes and discuss how basic block size affects their optimisation technique. Clausen et al. in [5] consider a number of techniques for the compression of Java Bytecode, and one such technique documented by Clausen et al. is basic block compression. They suggest the use of stream compression of basic blocks but recognise that the gains achievable are related to basic block size.

Gagnon in [10] introduces a technique for applying inline-threading to Java bytecode, a technique based upon the elimination of dispatch over-heads within basic blocks. Berndal et al. in [2] present a natural extension of inline-threading. Their technique dynamically collects traces of frequently occurring basic block sequences, and aims to eliminate dispatch code between inlined basic blocks.

Quantitative approaches to the analysis of dynamic instruction frequencies within benchmark applications as a way of modelling such programs have been undertaken within a number of studies [7, 4, 12, 14, 22, 8, 26, 27, 18, 15, 24]. In particular [7, 4, 12, 14, 22, 8, 26, 27] have profiled Java applications at the granularity of Java instructions, while [18, 15, 24] focus on sequences of instructions known as n-grams. These studies in part, have aided the work of Power and O'Donoghue in [19], in considering the viability of super instruction implementation within the Java Virtual Machine's instruction set. Dujmovic in [4] presents results on the dynamic timing of Java instruction bytecodes.

Casey et al. in [3] present a profiling interpreter generator that identifies superinstructions within Java bytecode. This interpreter also creates the code for superinstruction implementation, from the base instruction definitions. The work of Stephenson et al. in [25] considers the effects of Java instruction despecialisation, which is a technique that replaces specialised instructions such as iload_0 with their generic counterpart iload.

Dujmovic in [9] presents a program difference model for examining the difference/similarity between benchmark applications. A similar approach is undertaken by Horgan et al. in [14], who also present the effects compiler choice can have on their difference metric. Gregg et al. in [6] perform a method-level analysis of Java Grande and SPECjvm98 benchmark applications. In particular, they record the nature of method call sites and targets within these applications, and examine the polymorphicity of virtual method calls.

3. THE BENCHMARK SUITE

Our findings in this paper are based on the analysis of seven benchmark applications chosen from the SPEC JVM98 benchmark suite [23]. The SPEC suite was developed as a rigorous Java benchmark that would reflect the real-world behaviour of Java applications. The benchmark applications analysed are:

_201_compress: A Java implementation of the Modified Lempel-Zif file compression algorithm. The algorithm searches for frequently occurring substings and replaces them with variable size code. Real text files are passed to the Java LZW compressor.

_202_jess: Jess is a Java application based on the CLIPS expert shell system. The CLIPS system was developed by NASA in an attempt to provide decision-making freedom to planetary ground rovers.

_205_raytrace: This is a raytrace that works on a scene representing a dinosaur.

_209_db: Database test application that reads a one megabyte file into memory. This file consists of names, addresses and phone numbers. The application performs a number of database functions on the data set, such as additions, deletions and sorts. _209_db is the only application in the SPECjvm98 suite that is not based on a real-world application.

_213_javac: The Java compiler from Sun Microsystem's Java Development Kit (JDK) 1.0.2. A set of source files are compiled by the compiler.

_222_mpegaudio: This application decompresses a 4MB stream of audio files that comply with the ISO MPEG Layer-3 audio specification.

_228_jack: Jack is a Java parser generator based on the Purdue Compiler Construction Tool Set (PCCTS).

3.1 Methodology

In our experiments we use seven of the SPECjvm98 benchmark applications to study the static and dynamic structure of a Java application's basic blocks. The benchmark programs can be run with one of three data set sizes s1, s10 and s100. All experiments conducted were performed using a data set size of s100. All benchmark runs were carried out on a Dell Optiplex GX280 containing a 3GHz Intel Pentium IV processor with 1 Gb of RAM running the Fedora Core 4 distribution of GNU/Linux.

The Sun Java 2 Runtime Environment, Standard Edition (build 1.5.0_04-b05) was used to run all benchmarks and

the JVM was run in interpretor mode. To capture the basic blocks of an application, small probes were inserted within each basic block, and a trace of the execution of each basic block was then recorded. We did not record basic blocks within Java library code nor the basic blocks of SPECjvm98 application starter code. Java source code from the Gretel Residual Test Coverage Suite [21] that calculates basic blocks was incorporated into our experimental setup.

4. DEFINING A BASIC BLOCK

As outlined previously, we define a basic block within a Java method code to be a sequence of Java bytecode instructions, such that if the sequence's first instruction is executed by the Java Virtual Machine then all other instructions in the sequence are executed sequentially, if no runtime errors or exceptions occur. We also note that the instructions invokevirtual, invokeinterface, invokespecial and invokestatic cause control to change to the invoked method, but for the purposes of this paper these particular instruction are not considered to end a basic block.

Program 1 shows the source code of a simple Java class. This class was compiled using Sun's javac compiler and the resulting class file was disassembled using Sun's javap disassembler. The results from disassembling the code are shown in Program 2. The output from disassembling the code is annotated with labels: b1, b2, ..., b8 to identify the basic blocks within the instruction trace.

One possible way to identify the basic blocks contained in a method's code, is to draw a control flow graph of the instructions. Figure 1 shows the resulting control flow graph of the opcodes contained within the main() method of Program 2. We consider a basic block to begin at the first instruction of a method and end when an opcode in the control flow graph has more than one outgoing edge, or begin with an opcode that has more than one incoming edge and end with an opcode with more than one outgoing edge. Figure 1 shows all the basic blocks in the control flow graph.

Program 1 A simple program in Java.

```
public class Simple{
  public static void main(String[]args){
    int k = 0;
    int i = 0;
    while(k < 10){
      k = k + 2;
    }
    i++;
    while(k < 20){
      k = k + 2;
    }
  }
}
```

5. BASIC BLOCK STATIC AND DYNAMIC SIZE ANALYSIS

In this section we present our findings on the static and dynamic distribution of basic block sizes, across seven of the SPECjvm98 benchmark applications.

Program 2 Result of running javap disassembler on class file represented by code segment Program 1, also shown are the basic blocks contained within this bytecode trace.

```
public Test();
  public static void main(java.lang.String[]);

   Pos    Opcode      Operand        Basic block

    0:   iconst_0                        b2
    1:   istore_1                        b2
    2:   iconst_0                        b2
    3:   istore_2                        b2
    4:   iload_1                         b3
    5:   bipush          10              b3
    7:   if_icmpge       17              b3
   10:   iload_1                         b4
   11:   iconst_2                        b4
   12:   iadd                            b4
   13:   istore_1                        b4
   14:   goto            4               b4
   17:   iinc            2, 1            b5
   20:   iload_1                         b6
   21:   bipush          20              b6
   23:   if_icmpge                       b6
   26:   iload_1                         b7
   27:   iconst_2                        b7
   28:   iadd                            b7
   29:   istore_1                        b7
   30:   goto            20              b7
   33:   return                          b8
```

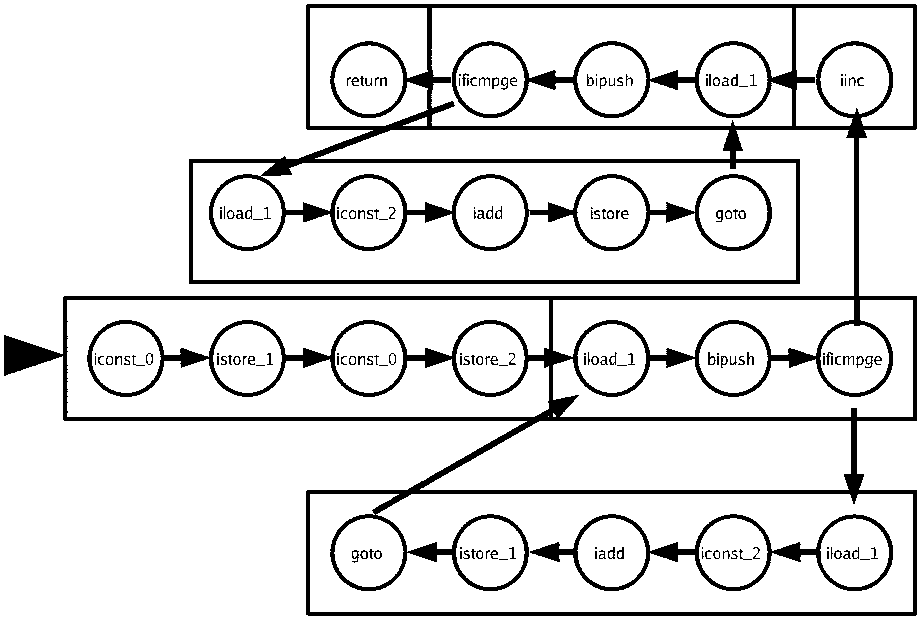

Figure 1: The control flow diagram shows the basic blocks representing code segment Program 1.

Table 1 shows the total number of basic blocks contained statically within SPECjvm98 application code and the number of basic blocks dynamically executed from these applications by the JVM.

Table 2 shows the percentage of total number of basic blocks in a SPECjvm98 application, that contain less than 5, 10, 15 and 20 instructions and also the percentage of basic blocks that contain at least 20 instructions. These results are based on the static analysis of basic blocks within the benchmark applications. For example we can see that 41.88% of basic blocks contained within _201_compress are composed of at most four instructions, and on average 48.40% of basic blocks within SPECjvm98 applications contain no more than four instructions. Our results tend to agree with those of Maierhofer and Ertl in [17] who present their findings on basic block sizes, obtained from the analysis of the JDK class library. They conclude that 50% of basic blocks contain no more than four instructions with 85% containing at most ten instructions.

Table 3 shows a similar analysis to that presented in Table 2 but considers those blocks that are dynamically executed by the JVM. We can see from these results that the percentages of dynamically executed basic blocks of a particular length seem to correlate with those percentages exhibited for the static analysis, and would merit further analysis.

Table 4 records the mean number of instructions $\overline{x}$, $\overline{y}$ composing static and dynamic basic blocks and also their associated standard deviations s_x and s_y. For example, the static basic blocks within the SPECjvm98 application _222_mpegaudio have a mean instruction count of 19.84 and standard deviation of 125.73 instructions. In considering the statistics presented in Table 4, Figure 2 shows the frequency distribution of basic block sizes across all seven benchmark applications. We also include a cumulative count of basic blocks containing at least 21 instructions. Figure 2 shows that basic block sizes of less than 21 instructions seem to follow a Poisson distribution. We can see from these figures that the accumulation of basic blocks containing at least 21 instructions occur quite frequently within the applications analysed statically and seem to contribute to the recorded deviations in static block sizes as presented in Table 4. For example, our analysis has identified that for the SPECjvm98 application _201_compress 18 basic blocks account for 9.42% of all static blocks and these have a mean instruction count of 31.9 instructions one of which contains 82 instructions. However the SPECjvm98 application _213_javac contains 34 basic blocks accounting for 3.60% of all static blocks, and of these two contain 895 and 1623 instructions. The application _222_mpegaudio contains four basic blocks with more than 1000 instructions, the largest basic block having 2831 instructions. In summation, large basic blocks seem to contribute to the high standard deviations recorded.

Dynamically we record considerably different results with the exception of _222_mpegaudio. With the exception of _222_mpegaudio the results recorded in Table 4 show that the standard deviations are very close to the mean values recorded, four benchmark applications recording a standard deviation less than their mean and two recording a standard deviation slightly greater than their mean. These results seem to indicate that although large blocks tend to skew the static block size statistics, dynamically these blocks are not executed very frequently. However _222_mpegaudio has a large standard deviation, this seems to be accounted for by one particular basic block containing 130 instructions that dynamically represents 5.09% of total execution.

	Number of Basic Blocks	
Spec Program	**Static**	**Dynamic**
_201_compress	191	122+e07
_202_jess	2510	30+e07
_205_raytrace	597	36+e07
_209_db	246	11+e07
_213_javac	5853	17+e07
_222_mpegaudio	910	66+e07
_228_jack	2150	3.77+e07

Table 1: Number of static and dynamic basic blocks in SPECjvm98 applications.

	Number of Instructions				
Spec Program	**< 5**	**< 10**	**< 15**	**< 20**	**> 19**
_201_compress	41.88	68.59	83.77	90.58	9.42
_202_jess	46.53	78.37	91.35	95.22	4.78
_205_raytrace	46.23	69.85	83.25	87.10	12.90
_209_db	56.91	82.52	93.90	95.93	4.07
_213_javac	46.68	80.28	93.18	96.40	3.60
_222_mpegaudio	49.01	75.93	85.71	90.33	9.67
_228_jack	51.58	72.74	83.44	96.42	3.58
average	*48.40*	*75.46*	*87.80*	*93.14*	*6.86*

Table 2: Number of instructions composing static basic blocks from SPECjvm98 applications.

	Number of Instructions				
Spec Program	**< 5**	**< 10**	**< 15**	**< 20**	**> 19**
_201_compress	31.17	61.62	79.16	83.82	16.18
_202_jess	59.85	89.35	96.21	98.69	1.31
_205_raytrace	68.80	83.19	91.43	93.69	6.31
_209_db	50.96	69.35	71.92	71.92	28.08
_213_javac	54.44	86.17	94.00	99.43	0.57
_222_mpegaudio	53.21	80.33	83.31	88.24	11.76
_228_jack	45.71	78.66	94.02	97.74	2.26
average	*52.02*	*78.38*	*87.15*	*90.50*	*9.50*

Table 3: Number of instructions composing dynamic basic blocks from SPECjvm98 applications.

	Static		Dynamic	
Spec Program	$\overline{\mathbf{x}}$	$\mathbf{s_x}$	$\overline{\mathbf{y}}$	$\mathbf{s_y}$
_201_compress	8.77	9.75	10.22	9.01
_202_jess	6.89	6.62	5.13	4.24
_205_raytrace	10.63	27.12	5.82	6.37
_209_db	6.11	6.26	9.40	9.76
_213_javac	6.77	24.87	5.36	4.17
_222_mpegaudio	19.84	125.73	16.52	59.92
_228_jack	8.31	47.21	6.13	5.86
average	*52.02*	*78.38*	*87.15*	*90.50*

Table 4: Mean number of instructions $\overline{x}$, $\overline{y}$ composing static and dynamic basic blocks and there respective standard deviations s_x and s_y.

From Figure 2 we note that basic blocks containing three instructions are the most frequent size of basic block found within SPECjvm98 applications.

6. FREQUENTLY EXECUTED BASIC BLOCKS

Tables 5 through 11 give the frequencies and instructions for the top 5 most frequently executed basic blocks from each of the SPECjvm98 applications analysed. For each table the first column indicates the number of Java instructions contained in the basic block, the second column enumerates the basic block bytecode instructions, the third column lists the total frequency of occurrence, and the fourth column expresses the total basic block frequency as a percentage of total basic blocks executed within that particular application.

From Tables 5 through 11 we note that basic blocks containing no more than five instructions account for approximately 77% of all the top 5 repeatedly ocurring basic blocks recorded. In particular the basic block containing one instruction namely the 'iinc' instruction appears within the top 5 basic blocks for four of the applications analysed, and is recorded as the most frequently executed block within _228_jack. Table 10 records the largest basic block, contaning 130 instructions. We see from this particular block that floating point instructions represent the most frequent type of instruction executed, with object referencing accounting for the second most frequenct type of instruction occuring within this block. The abundance of floating point operations seems to be indicative of mpegaudio's encoding and decoding operations.

Our results also show that the top 5 most frequently executed basic blocks on average across all applications account for approximately 40% of total basic blocks executed. _205_raytrace exhibits the greatest percentage, with a percentage of 70.47%, and one basic block "aload_0 getfield freturn", representing 42.40% of execution. Figure 3 shows the top 5 basic blocks from each application, and the total percentage of execution these represent.

Figure 4 shows the distribution of the remaining basic blocks from a SPECjvm98 application that comprise an execution of less than 1% and the percentage of static basic blocks. The results presented so far indicate that a small percentage of basic blocks account for the vast amount of execution, while a large number of basic blocks are executed very infrequently. This distribution of frequencies seems to indicate the presence of a power law [1] relationship between dynamic block execution and static block frequency.

On close examination of the instructions that compose the basic blocks shown in tables 5 through 11 we see that the bigram "aload_0 getfield" identified by O'Donoghue et al. in [18], occurs in 14 of the 35 basic blocks recorded. In considering the analysis undertaken by Leddy in [15] we find that the trigram "aload_0 getfield iload_1" also is present. We also note the presence of basic blocks that perform similar tasks but use different instruction. For example, the two basic blocks "iload_2 aload_0 getfield invokevirtual if_icmplt" and "iload aload_0 getfield invokevirtual if_icmplt" occurring in _228_jack only differ in their first instruction. These basic

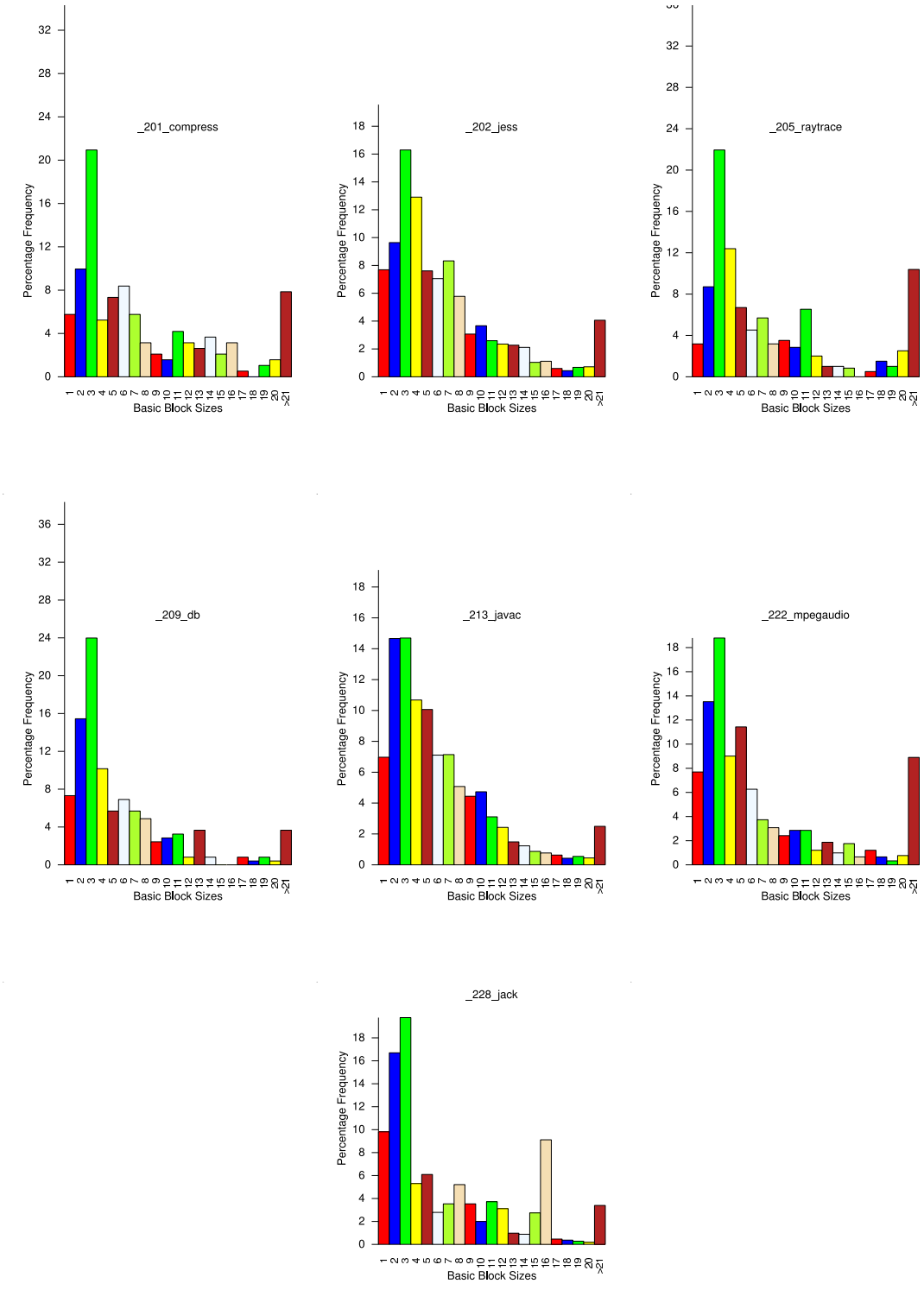

Figure 2: These graphs show the frequency distribution of basic block sizes, in seven of the SPECjvm98 benchmark applications.

Compress

Size	Basic Block Instructions	Frequency Total	%
6	ILOAD_2 ILOAD ISUB DUP ISTORE_2 IFGE	8.5e+07	6.98
11	ALOAD_0 GETFIELD ASTORE ALOAD GETFIELD ILOAD_2 IALOAD ISTORE ILOAD ILOAD_1 IF_ICMPNE	8.5e+07	6.98
2	ILOAD IFGT	7.5e+07	6.16
9	ALOAD_0 GETFIELD ILOAD_1 SALOAD BIPUSH ISHL BIPUSH IUSHR IRETURN	7.44e+07	6.10
12	ALOAD_0 GETFIELD ALOAD_0 DUP GETFIELD DUP_X1 ICONST_1 IADD PUTFIELD ILOAD_1 BASTORE RETURN	6.6e+07	5.38
	Total	*3.85e+08*	*31.6*

Table 5: Top five most frequently executed basic blocks from _201_compress

Jess

Size	Basic Block Instructions	Frequency Total	%
5	ALOAD_0 GETFIELD ILOAD_1 AALOAD ARETURN	2.7e+07	8.87
1	IINC	2.5e+07	8.28
3	ALOAD_0 ALOAD_1 IF_ACMPNE	2.4e+07	7.81
4	ILOAD_3 ALOAD_0 GETFIELD IF_ICMPLT	1.8e+07	6.14
2	ICONST_0 IRETURN	1.5e+07	4.87
	Total	*8.2e+07*	*35.97*

Table 6: Top five most frequently executed basic blocks from _202_jess

Raytrace

Size	Basic Block Instructions	Frequency Total	%
3	ALOAD_0 GETFIELD FRETURN	1.5e+08	42.40
3	ALOAD_0 GETFIELD ARETURN	4.9e+07	13.46
5	ALOAD_0 GETFIELD ILOAD_1 AALOAD ARETURN	3.3e+07	9.04
35	ALOAD_0 FLOAD_3 ALOAD_1 INVOKEVIRTUAL FMUL FLOAD ALOAD_2 INVOKEVIRTUAL FMUL FADD PUTFIELD ALOAD_0 FLOAD_3 ALOAD_1 INVOKEVIRTUAL FMUL FLOAD ALOAD_2 INVOKEVIRTUAL FMUL FADD PUTFIELD ALOAD_0 FLOAD_3 ALOAD_1 INVOKEVIRTUAL FMUL FLOAD ALOAD_2 INVOKEVIRTUAL FMUL FADD PUTFIELD ALOAD_0 ARETURN	6.0e+06	1.68
4	FLOAD FLOAD FCMPL IFLE	5.2e+06	1.43
	Total	*2.4e+08*	*68.01*

Table 7: Top five most frequently executed basic blocks from _205_raytrace

Db

Size	Basic Block Instructions	Frequency Total	%
2	ILOAD_3 IFGE	2.3e+07	19.44
24	ALOAD_0 GETFIELD ILOAD_3 AALOAD GETFIELD ILOAD_1 INVOKEVIRTUAL CHECKCAST ASTORE ALOAD_0 GETFIELD ILOAD_3 ILOAD IADD AALOAD GETFIELD ILOAD_1 INVOKEVIRTUAL CHECKCAST ASTORE ALOAD ALOAD INVOKEVIRTUAL IFLE	2.3e+07	18.99
1	IINC	1.4e+07	11.34
3	ILOAD_2 ILOAD IF_ICMPLT	1.2e+07	10.35
5	ILOAD_2 ILOAD ISUB ISTORE_3 GOTO	1.2e+07	10.35
	Total	*8.4e+07*	*70.47*

Table 8: Top five most frequently executed basic blocks from _209_db

Javac

Size	Basic Block Instructions	Frequency Total	%
16	ALOAD_0 ALOAD_0 GETFIELD PUTFIELD ALOAD_0 DUP GETFIELD ICONST_1 IADD PUTFIELD ALOAD_0 GETFIELD ISTORE_1 ILOAD_1 ICONST_M1 IF_ICMPNE	8.0e+06	4.70
2	ILOAD_1 LOOKUPSWITCH	8.0e+06	4.70
5	ALOAD_0 GETFIELD INVOKEVIRTUAL ISTORE_1 GOTO	7.9e+06	4.68
2	ILOAD_1 IRETURN	7.8e+06	4.62
3	ALOAD_0 GETFIELD ARETURN	7.3e+06	4.31
	Total	*3.9e+07*	*23.01*

Table 9: Top five most frequently executed basic blocks from _213_javac

Mpegaudio

Size	Basic Block Instructions	Frequency Total	%
3	ILOAD BIPUSH IF_ICMPLT	5.7e+07	8.55
130	GETSTATIC ILOAD IINC AALOAD ASTORE FLOAD_3 ALOAD_0 GETFIELD ILOAD AALOAD ILOAD BIPUSH IADD FALOAD ALOAD ICONST_0 FALOAD FMUL FADD FSTORE_3 FLOAD ALOAD_0 GETFIELD ILOAD AALOAD ILOAD BIPUSH IADD FALOAD ALOAD ICONST_1 FALOAD FMUL FADD FSTORE FLOAD ALOAD_0 GETFIELD ILOAD AALOAD ILOAD BIPUSH IADD FALOAD ALOAD ICONST_0 FALOAD FMUL FADD FSTORE FLOAD ALOAD_0 GETFIELD ILOAD AALOAD ILOAD BIPUSH IADD FALOAD ALOAD ICONST_1 FALOAD FMUL FADD FSTORE ILOAD ICONST_1 IADD BIPUSH IAND ISTORE FLOAD_3 ALOAD_0 GETFIELD ILOAD AALOAD ILOAD FALOAD ALOAD ICONST_2 FALOAD FMUL FADD FSTORE_3 FLOAD ALOAD_0 GETFIELD ILOAD AALOAD ILOAD FALOAD ALOAD ICONST_3 FALOAD FMUL FADD FSTORE FLOAD ALOAD_0 GETFIELD ILOAD AALOAD ILOAD FALOAD ALOAD ICONST_2 FALOAD FMUL FADD FSTORE FLOAD ALOAD_0 GETFIELD ILOAD AALOAD ILOAD FALOAD ALOAD ICONST_3 FALOAD FMUL FADD FSTORE ILOAD ICONST_1 IADD BIPUSH IAND ISTORE IINC	3.4e+07	5.09
1	IINC	2.3e+07	3.54
3	ILOAD ILOAD_3 IF_ICMPLT	2.1e+07	3.13
3	ILOAD_3 BIPUSH IF_ICMPLT	2.1e+07	3.10
	Total	*1.6e+08*	*23.41*

Table 10: Top five most frequently executed basic blocks from _222_mpegaudio

Jack

Size	Basic Block Instructions	Frequency Total	%
1	IINC	3.1e+06	8.26
3	ALOAD INVOKEINTERFACE IFNE	1.9e+06	5.00
5	ILOAD ALOAD_0 GETFIELD INVOKEVIRTUAL IF_ICMPLT	1.7e+06	4.40
5	ILOAD_2 ALOAD_0 GETFIELD INVOKEVIRTUAL IF_ICMPLT	1.5e+06	3.91
2	ILOAD IRETURN	1.3e+06	3.60
	Total	*9.5e+06*	*25.17*

Table 11: Top five most frequently executed basic blocks from _228_jack

Spec Program	no. of basic blocks with an execution greater 1%	static % of unique basic blocks
_201_compress	21	13.46
_202_jess	29	2.47
_205_raytrace	11	3.10
_209_db	16	9.20
_213_javac	19	0.74
_222_mpegaudio	30	6.00
_228_jack	25	4.35
Average	*21.5*	*5.6*

Table 12: The number of basic blocks that represent at least 1% of execution, and their percentage of the total number of unique static blocks

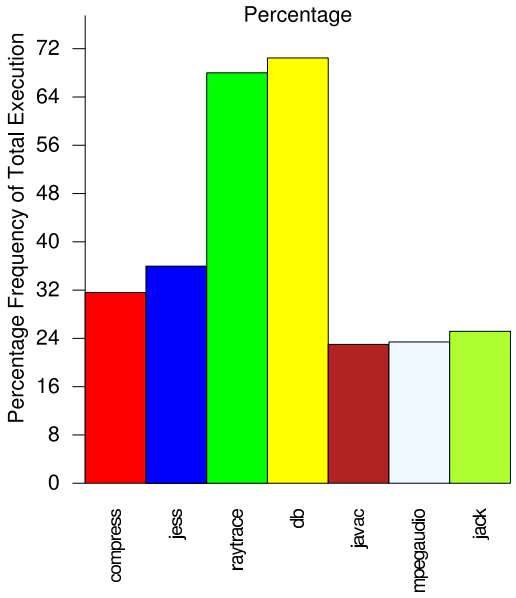

Figure 3: This chart shows for each SPECjvm98 application the percentage of total basic blocks executed represented by the top 5 most frequent basic blocks.

blocks are categorically equivalent, both iload_2 and iload performing a push operation of an integer constant onto the JVM stack.

Table 12 records the number of basic blocks that individually account for a dynamic execution of more than 1% of total execution. Column 1 indicates the SPECjvm98 application, column 2 records the number of basic blocks dynamically executed that individually account for more than 1% of total execution and column 3 indicates the percentage of unique static basic blocks that these represent. These results indicate that on average across all applications, 21.5 basic blocks dynamically executed represent 5.6% of an application's static basic blocks. These results would seem to agree with the suggestion put forward by Proebsting in [20] for C programs, that a maximum of 20 superoperators would suffice to get full advantage of the technique of superinstruction implementation.

In considering the results presented in Table 12 we see that on average 21.5 basic blocks have a dynamic execution of more than 1%, excluding the top 5 most frequenctly executed basic blocks, at least another 15% of total dynamic execution is accounted for by the next 15 most frequenctly executed basic blocks. This results in an average of 55% of execution being accounted for by the top 21.5 basic blocks, representing an average of 5.6% of the static basic blocks within an application.

The technique used here to identify basic blocks is based on the code supplied with the Gretel Residual Test Coverage Tool [21]. This tool does not recognise method invocation instructions such as "invokevirtual", "invokeinterface", "invokestatic" and "invokespecial" as ending a basic block. We believe that it is reasonable to consider these particular instruction types as causing such a change and thus would effect the identification of basic blocks within application code.

In considering the later point, we note the effect this consideration would have on basic block identification. The applications _205_raytrace (Table 7), _209_db (Table 8), _213_javac (Table 9) and _228_jack (Table 9) all record method invocation instructions within their top 5 most frequently occurring basic blocks. As these particular basic blocks account for the most frequently occurring blocks, the result of decomposing these into smaller basic blocks would cause a greater number of smaller basic blocks to be registered as most frequently occurring blocks.

7. BASIC BLOCKS AS A REPRESENTATIVE SAMPLE OF BENCHMARK APPLICATION WORKLOADS

In this section we present our findings from an initial study into the viability of sampling an application's basic block trace as a way of producing a representative sample of an application's behaviour. The program difference model

$$d(A,B) = \frac{1}{2}\sum_{i=1}^{N} |p_i^{(A)} - p_i^{(B)}| \quad (1)$$

proposed by Dujmovic in [9] was used to assign a similarity value to basic block samples taken from an application. The frequency f_i, i = 0,..., 202 of each instruction was calculated for each SPECjvm98 application 'A', and the frequency of each instruction occurring in the sample 'B' of basic blocks was also calculated. The relative frequencies p_i^A AND p_i^B of each instruction executed in the application and sample respectively, was calculated using $\frac{f_i}{\sum_{i=0}^{202} f_i}$.

We define the workload of an application to mean the frequency of instructions dynamically executed. The workload difference was calculated using the workload of the entire application, and the workload associated with the most frequently executed basic blocks. We examined the differences in workload for 10%, 20%, ..., 90% of frequently executed basic blocks.

Table 13 shows the results of our investigation where a value of 0.0 indicates identical workloads, and 1.0 indicating totally different workloads. For example the workload associated with the basic blocks of _201_compress is nearly identical to the workload associated with the sampled basic blocks that account for 60% of _201_compress, having a similarity value of 0.11.

Considering all applications analysed, the results shown in Figure 13 indicate that the top 80% of dynamically executed basic blocks in each application only differ from the workloads of the full application by at most 21%. In addition, in six of the seven applications the top 70% of basic blocks differ by at most 20%.

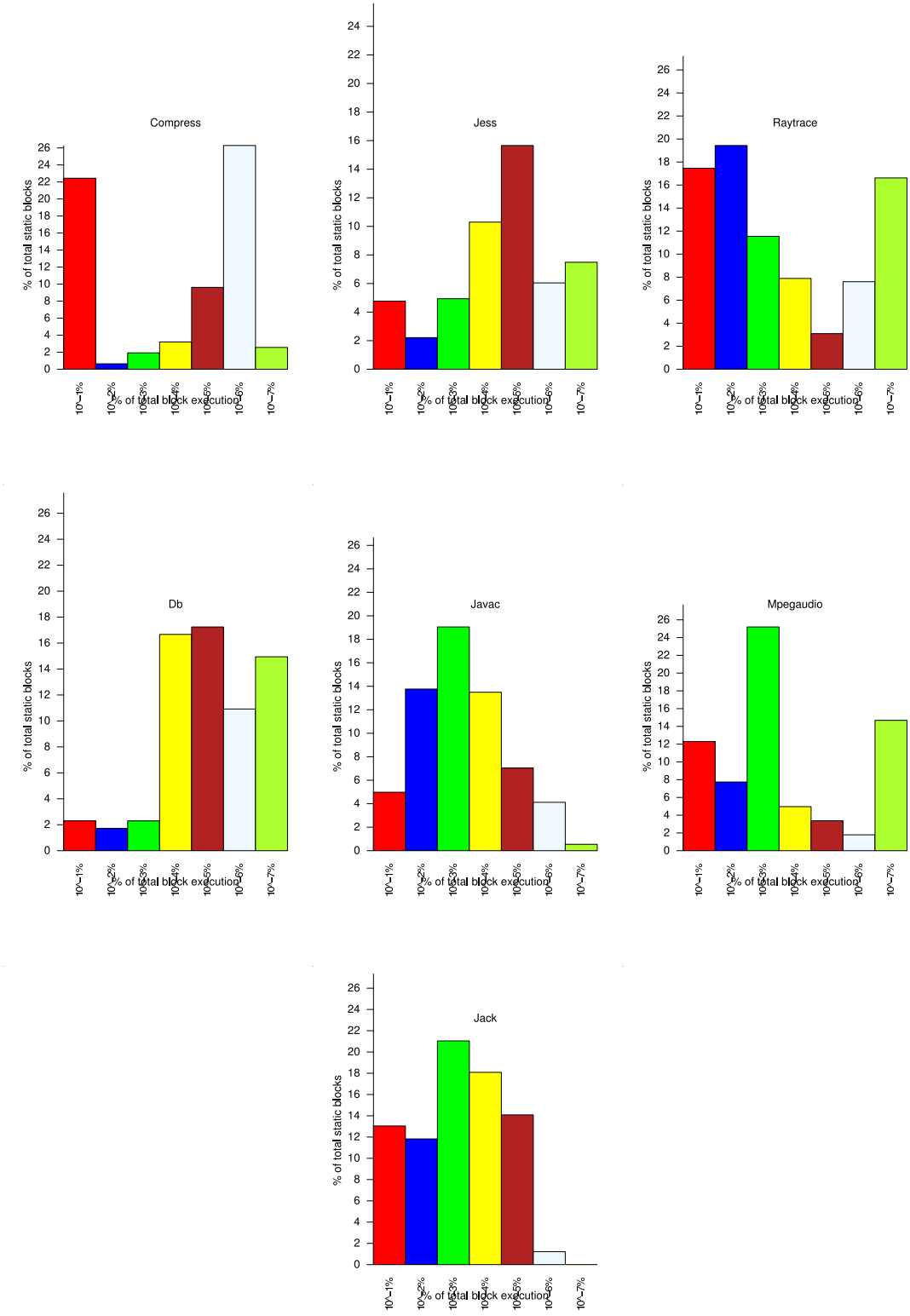

Figure 4: The above histograms show the percentage of static basic blocks and their percentage frequency of execution.

	Basic Blocks Sampled								
Spec Program	**10%**	**20%**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**
_201_compress	0.46	0.47	0.27	0.25	0.17	0.11	0.09	0.06	0.02
_202_jess	0.63	0.58	0.49	0.44	0.37	0.25	0.20	0.12	0.05
_205_raytrace	0.61	0.61	0.61	0.61	0.57	0.52	0.28	0.21	0.10
_209_db	0.87	0.23	0.23	0.21	0.18	0.18	0.14	0.05	0.03
_213_javac	0.38	0.35	0.33	0.31	0.25	0.17	0.14	0.08	0.04
_222_mpegaudio	0.25	0.25	0.23	0.19	0.17	0.15	0.12	0.11	0.11
_228_jack	0.90	0.49	0.37	0.30	0.28	0.19	0.14	0.08	0.03

Table 13: Similarity between SPECjvm98 applications and Sampled Basic Blocks

Spec Program	$\overline{x}$	s_x	$\overline{y}$	s_y	r
_201_compress	0.64	0.39	0.74	1.42	0.11
_202_jess	0.09	0.21	0.16	0.57	0.40
_205_raytrace	0.28	0.36	0.33	2.41	0.65
_209_db	0.58	0.51	0.76	2.60	0.10
_213_javac	0.04	0.10	0.06	0.26	0.38
_222_mpegaudio	0.20	0.34	0.26	0.68	0.27
_228_jack	0.17	0.49	0.21	0.65	0.11

Table 14: Correlation between static basic block frequencies and dynamic basic block frequencies

8. RELATING STATIC AND DYNAMIC BASIC BLOCK FREQUENCIES

In this section we explore whether there exists a linear relationship between static and dynamic basic block frequencies. A similar study was undertaken by Dowling et al. in [8] but was conducted at the individual instruction level, and Dowling et al. concluded that no overall linear relationship exits between static and dynamic instruction frequencies.

Pearson's correlation coefficient was used, and is given by:

$$r = \frac{\sum(x-\overline{x})(y-\overline{y})}{(n-1)s_x s_y} \quad (2)$$

In equation 2 the static and dynamic basic block frequencies are represented for two sets of data of size n by the random variables x and y, having means $\overline{x}$ and $\overline{y}$ and standard deviations s_x and s_y.

In our calculation we note that the static basic block frequencies recorded accounted for all basic blocks comprising a particular application, whereas it was possible that the dynamic frequencies did not include all such basic blocks. In such a case a frequency of zero was returned.

Our results from correlating static and dynamic basic block frequencies are shown in Table 14. Pearson's correlation coefficient returns a value in the range of -1 to 1, and $\mid r \mid > 0.8$ indicates a strong linear corelation, $0.5 \leq \mid r \mid \leq 0.8$ denoting a moderate linear correlation and values of $\mid r \mid \leq 0.5$ denoting a weak linear correlation. We note from our results that no application exhibits a strong linear correlation between static and dynamic basic block frequencies and only _205_raytrace shows a moderate correlation.

9. CONCLUSION AND FUTURE WORK

In this paper we have presented the results of an analysis of basic block sizes, occurring both statically and dynamically. We conclude that for SPECjvm98 applications both statically and dynamically occurring basic blocks are small in size, with 90% containing less than 20 instructions both statically and dynamically, and approximatly 50% of SPECjvm98 application basic blocks containing less than 5 instructions, a result that tends to agree with those put forward by Maierhofer and Ertl in [17] and would seem to suggest a pattern accross all Java applications.

From the dynamic analysis of basic block frequencies, we conclude that a small number of static basic blocks, account for a large proportion of dynamic execution, a result that would seem to suggest the presence of a power law distribution. Also we see that approximately 77% of the most frequently occurring basic blocks contain no more than 5 instructions. We also note that the bigram "aload_0 getfield" is a common sequence ocurring in 14 of the 35 basic blocks reported. In particular it starts a basic block very frequently. We also conclude from our results that the choice of 20 superinstructions as suggested by Proebsting seems to be a logical threshold, that will afford maximum benefit to the technique of superinstruction implementation within a JVM.

Our results also show that the top 80% of the most frequently executed basic blocks differ from the workload of the total application by as little as 12%. We also conclude that there does not exist a linear correlation between static block frequencies and dynamic frequencies, a result that was shown by Dowling in [8] to hold for instruction frequencies.

It is proposed in the future to examine the effects of Java bytecode categorisation on basic block frequency distribution and also to explore the concept of statistically improbable basic blocks as a way identifying particular types of Java applications.

10. REFERENCES

[1] L. Adamic. Zipf, power-laws, and pareto - a ranking tutorial. http://www.hpl.hp.com/research/idl/papers/ranking/ranking.html.

[2] Marc Berndl and Laurie Hendren. Dynamic profiling and trace cache generation for a Java Virtual Machine. Technical Report No. 2002-8, San Francisco State University, Department of Computer Science,

	Number of basic blocks			
Benchmark	**Total**	**Unique**	**Not Executed**	**Executed**
_201_compress	191	156	22	135
_202_jess	2510	1175	542	633
_205_raytrace	597	355	47	308
_209_db	246	174	43	131
_213_javac	5853	2572	933	1639
_222_mpegaudio	910	504	121	383
_228_jack	2150	575	94	481

Table 15: Total number of basic blocks within each benchmark, number of unique basic blocks, number of basic blocks not executed and the total of unique basic blocks executed.

August 2002.

[3] Kevin Casey, David Gregg, and Anton Ertl. Towards superinstructions for Java interpreters. In *7th International Workshop on Software and Compilers for Embedded Systems*, Vienna, Austria, September 24-26 2003.

[4] Herder C.C and Dujmovic J.J. Frequency analysis and timing of Java bytecodes. Technical Report SFSU-CS-TR-00.02, San Francisco State University, Department of Computer Science, January 2000.

[5] Lars Rder Clausen, Ulrik Pagh Schultz, Charles Consel, and Gilles Muller. Java bytecode compression for low-end embedded systems. *ACM Transactions on Programming Languages and Systems*, 22(3):471–489, May 2000.

[6] J. Power D. Gregg and J.T. Waldron. A method-level comparison of the Java Grande and SPECjvm98 benchmark suites. *Journal of Concurrency and Computation Practice and Experience*, 17(7-8), June-July 2005.

[7] Charles Daly, Jane Horgan, James F. Power, and John T. Waldron. Platform independent dynamic Java Virtual Machine analysis: the Java Grande Forum Benchmark Suite. In *Joint ACM Java Grande - ISCOPE Conference*, pages 106–115, Stanford, CA, USA, June 2001.

[8] T. Dowling, James F. Power, and J. T. Waldron. Relating static and dynamic measurements for the Java virtual machine instruction set. In *Symposium on Mathematical Methods and Computational Techniques in Electronic Engineering*, Athens, Greece, December 29-31 2001.

[9] J.J. Dujmovic. Universal benchmark suites. In *MASCOTS'99 Conference Proceedings, IEEE Computer Society Press*, pages 197–205, 1999.

[10] Etienne Gagnon. *A Portable Research Framework for the Execution of Java Bytecode*. PhD thesis, McGill University, 2002.

[11] J. Gosling and B. Joy *The Java Language Specification* Second Edition. *The Java Language Specification*. McGraw Hill, 1998.

[12] D. Gregg, James F. Power, and J. T. Waldron. Benchmarking the Java virtual architecture - the SPEC JVM98 benchmark suite. In N. Vijaykrishnan and M. Wolczko, editors, *Java Microarchitectures*, chapter 1, pages 1–18. Kluwer Academic, 2002.

[13] David Gregg, M. Anton Ertl, and John Waldron. The common case in Forth programs. In *EuroForth 2001 Conference Proceedings*, pages 63–70, 2001.

[14] J. Horgan, J. Power, and J. Waldron. Measurement and analysis of runtime profiling data for Java programs. In *First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 122–130, Florence, Italy, November 10 2001.

[15] Aine Leddy. Dynamic bytecode analysis using n-grams. 2001.

[16] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1996.

[17] Martin Maierhofer and M. Anton Ertl. Local stack allocation. In *CC '98: Proceedings of the 7th International Conference on Compiler Construction*, pages 189–203, London, UK, 1998. Springer-Verlag.

[18] D. O'Donoghue, A. Leddy, James F. Power, and J. T. Waldron. Bi-gram analysis of Java bytecode sequences. In *Second Workshop on Intermediate Representation Engineering for the Java Virtual Machine*, pages 187–192, Dublin, Ireland, June 13-14 2002.

[19] Diarmuid O'Donoghue and James F. Power. Identifying and evaluating a generic set of superinstructions for embedded Java programs. In *International Conference on Embedded Systems and Applications*, June 2004.

[20] Todd A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 322–332, New York, NY, USA, 1995.

[21] Jakarta Project. The gretel residual test coverage tool. http://www.jakarta.apache.org/bcel/projects.html.

[22] R. Radhakrishnan, N. Vijaykrishnan, L.K. John, A. Sivasubramaniam, J. Rubio, and J. Sabarinathan. Java runtime systems: Characterization and architectural implications. *IEEE Transactions on Computers*, 50(2):131–146, February 2001.

[23] SPEC. The specjvm98 benchmark suite. http://www.spec.org/osg/jvm98.

[24] Ben Stephenson and Wade Holst. A quantitative analysis of Java bytecode sequences. In *Proceedings of the 3rd International Symposium on Principles and Practice of Programming in Java*, pages 15–20. Trinity College Dublin, 2004.

[25] Ben Stephenson and Wade Holst. A quantitative analysis of the performance impact of specialized bytecodes in Java. In *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, pages 267–281. IBM Press, 2004.

[26] J. Waldron. Dynamic bytecode usage by object oriented Java programs. In *Technology of Object-Oriented Languages*, pages 384–393, Nancy, France, June 1999.

[27] J. Waldron, C. Daly, D. Gray, and J. Horgan. Comparison of factors influencing bytecode usage in the Java Virtual Machine. In *Second International Conference and Exhibition on the Practical Application of Java*, pages 315–327, Manchester, UK, April 2000.

[28] Maurer D. Wilhelm R. *Compiler Design*. Addison Wesley, 1995.