Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 220 (2008) 97–113
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Platform Independent Timing of Java Virtual Machine Bytecode Instructions

Jonathan M. Lambert[1] and James F. Power

Department of Computer Science, National University of Ireland Maynooth, Co. Kildare, Ireland

Abstract

The accurate measurement of the execution time of Java bytecode is one factor that is important in order to estimate the total execution time of a Java application running on a Java Virtual Machine. In this paper we document the difficulties and solutions for the accurate timing of Java bytecode. We also identify trends across the execution times recorded for all imperative Java bytecodes. These trends would suggest that knowing the execution times of a small subset of the Java bytecode instructions would be sufficient to model the execution times of the remainder. We first review a statistical approach for achieving high precision timing results for Java bytecode using low precision timers and then present a more suitable technique using homogeneous bytecode sequences for recording such information. We finally compare instruction execution times acquired using this platform independent technique against execution times recorded using the read time stamp counter assembly instruction. In particular our results show the existence of a strong linear correlation between both techniques.

Keywords: Java Virtual Machine, Bytecode Timing, Bytecode Sequences, RDTSC.

1 Introduction

The popularity and portability of the Java programming language has allowed Java technology to take hold across a range of devices such as embedded systems, smart card devices as well as the traditional role of Java on desktops and servers. Benchmarking programs in such a variety of environments raises the new research challenge of how to accurately and reliably measure Java programs in a platform-independent manner.

There are a number of existing approaches to timing Java programs at the bytecode level. Several special-purpose implementations of the Java Virtual Machine (JVM) have been constructed for research purposes and provide an interface that allows the extraction of profiling information [11,6,7]. More directly, using an open-source JVM such as Kaffe [25] permits the researcher to directly instrument the

[1] Address correspondence to J.M. Lambert at jonathan@cs.nuim.ie

doi:10.1016/j.entcs.2008.11.021

dispatch loop associated with instruction execution. Both of these approaches may be characterised as "white-box" in the context of the JVM, since they are tied to a particular JVM implementation, and require some specialised knowledge of that implementation.

An alternative approach is to treat the JVM as a "black-box", and use a standard, off-the-shelf JVM where the researcher does not have direct access to the underlying JVM source code. This has the advantage of permitting maximum flexibility in terms of the choice of platform and the range of programs which can be executed. It has the disadvantage that the researcher must rely on the standard Java library timing methods which are quite coarse-grained.

This paper addresses the basic question: *to what extent can we reliably predict the execution timings for JVM bytecode instructions at this kind of platform-independent level?* Thus, in this paper we present a platform independent technique for achieving reliable execution times for Java bytecode instructions running in an environment where the JVM source code is not available and we measure the accuracy of this technique.

The remainder of this paper is structured as follows. Section 2 documents background and related work undertaken in this field and Section 3 documents our experimental design and methodology. Section 4 presents and analyses our platform independent instruction timing results and Section 5 compares these results against results acquired using the RDTSC assembly instruction. Finally, Section 6 concludes the paper and identifies some future areas of research on this topic.

2 Background and Related Work

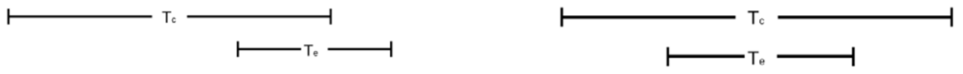
In this section we present the background of the instruction execution model, we introduce Java instructions into the model, and we discuss the tractability issues associated with our technique. We also present a synopsis of related work.

2.1 A binomial model of instruction execution

The measurement of the execution time of an event is analogous to the process of starting and stopping a stop watch. The event being timed is typically preceded by a call to a timing function and followed by a final call to the timing function.

One factor that can affect the accuracy of a timing technique is the resolution of the timing clock. The resolution of the clock used to time an event is defined as the smallest possible value, other than zero, of the difference between two successive observations of a clock's value. Thus a clock's resolution determines the shortest event that can be accurately measured with the clock. For example, a clock whose value changes every millisecond can only be used to time an event whose duration is at least a millisecond in length. Using a clock with a resolution greater than the duration of an event can result in one of two possible results depending on when the clock's values are observed with respect to the event being measured.

Figure 1 depicts the two possibilities of an event's location with respect to a clock advancing one time unit. Figure 1(a) shows an event whose duration T_e straddles



(a) T_e straddles the edge of T_c (b) T_e begins and ends within T_c

Fig. 1. A visualisation of two possible scenarios for the relative durations of the event to be timed T_e and the timing clock T_c where $T_e < T_c$.

the periodic advancement of the clock. In this instance observing the clock at the start and end of the event would show that one time unit had elapsed. In contrast, Figure 1(b) shows an event whose (identical) duration T_e falls entirely within the periodic advancement of the clock. In this instance observing the clock at the start and end of the event would return a difference of zero time units.

The timing of individual Java bytecode instructions is particularly susceptible to the above mentioned quantisation errors. Our measurements have shown that Java bytecode instructions execute within nanoseconds. Attempting to measure these instructions with a high degree of precision using standard Java library timing methods such as *System.currentTimeMillis* or *System.nanoTime* results in the quantisation errors masking their true execution times.

We can model this quantisation effect using standard statistical techniques to achieve high resolution timing in the presence of low resolution clocks. Lilja documents the use of Bernoulli trials to achieve high resolution timing [18]. Beilner presents a model of event timing and presents timing results using his technique for the times taken to pass a message between two processes [4]. Danzig et al. use the technique to develop a hardware micro-timer to time machine code executing on Sun 3 and Sun 4 workstation [9].

Modelling the above mentioned quantisation errors statistically involves considering the timing of an event as a Bernoulli trial [18,14]. The duration of the event being measured using a low resolution clock will either be 0 or 1 depending on whether the clock advanced a time unit during the observation of the start and finishing of the event. The outcome of this experiment is 1 with a probability of p if the clock advances while measuring the event, otherwise the outcome is 0 with a probability of $(1-p)$. If this experiment is repeated n times, the resulting distribution will approximate the binomial distribution [14]. After performing the Bernoulli trial n times, we find that the number of outcomes that produce 1 is m, then the ratio $\frac{m}{n}$ should approximate the ratio $\frac{T_e}{T_c}$ of the duration of the event to the clock period. Equating both ratios we can estimate the duration T_e of the event as shown in equation 1.

$$T_e = \frac{m}{n} T_c \tag{1}$$

We can then obtain a confidence interval for the average execution time $\overline{p}$ by evaluating the statistic for the confidence interval of a proportion as shown in equation 2. Here $z_{1-\frac{\alpha}{2}}$ represents the value of a standard unit normal distribution with an area of $1 - \frac{\alpha}{2}$ falling to the left [18].

$$\overline{p} \pm z_{1-\frac{\alpha}{2}} \sqrt{\frac{\overline{p}(1-\overline{p})}{n}} \tag{2}$$

A natural question arises from the above calculations, namely, how many times should the Bernoulli trial be performed? Equation 2 tells us that the true value p is within the interval $((1-\epsilon)\overline{p}, (1+\epsilon)\overline{p})$. Equating this interval with Equation 2 we get:

$$(1-\epsilon)\overline{p}, (1+\epsilon)\overline{p} = \overline{p} \pm z_{1-\frac{\alpha}{2}} \sqrt{\frac{\overline{p}(1-\overline{p})}{n}} \tag{3}$$

As $((1-\epsilon)\overline{p}, (1+\epsilon)\overline{p})$ forms a symmetric interval about $\overline{p}$ we can choose either bound to arrive at:

$$(1+\epsilon)\overline{p} = \overline{p} + z_{1-\frac{\alpha}{2}} \sqrt{\frac{\overline{p}(1-\overline{p})}{n}} \tag{4}$$

And finally solving for n gives

$$n = \frac{(z_{1-\frac{\alpha}{2}})^2 \, \overline{p} \, q}{(\epsilon\overline{p})^2} \tag{5}$$

Where n is the number of iterations required to measure a code segment of duration δ within an error margin of ϵ using a clock of resolution Δ seconds. Here $\overline{p} = \frac{\delta}{\Delta}$ and $q = (1-\overline{p})$.

From equation 5 we can calculate the number of Bernoulli experiments to be performed when timing a segment of code using a low resolution timer.

2.2 *Dealing with timer overhead*

Applying the above technique to the domain of Java bytecode instruction timing raises an important question: What event is being measured? Using the above technique the event being approximated using the Binomial model is the timer overhead plus the event's duration represented by T_{e+o} in equation 6. The execution time of the event, T_e, of interest is given by equation 6.

$$T_e = T_{e+o} - T_o \tag{6}$$

The confidence in the recorded value T_e is dependent on the number of significant digits associated with the results T_{e+o} and T_o. For example, consider timing an event whose duration is approximately 10^{-3} seconds and the timing of this event incurs an overhead penalty of approximately 10^{-1} seconds. To be confident with the value T_e we will need to estimate both T_{e+o} and T_o to three significant digits. Timing individual Java bytecode instructions whose duration is approximately in the order of a couple of nanoseconds using a clock with an estimated overhead of approximately 10 microseconds would require that we estimate both T_{e+o} and T_o to 4 significant digits so as to be confident with the result T_e. If we require the result, T_e, to a higher degree of precision then T_{e+o} and T_o would have to be estimated to more significant digits.

From equation 5, we see that the number of trials to be performed is dependent on the square of ϵ. In particular, small changes in ϵ can result in an unacceptable

$\alpha = 0.95$
$z_{95} = 1.96$
$\delta = 10^{-5}$
$\Delta = 10^{-3}$
$p = \frac{\delta}{\Delta} = 10^{-2}$
$q = 1 - p = 1 - 10^{-2}$

(a) Parameter settings

SD	$\epsilon\overline{p}$	n	Estimated Time
1	10^{-5}	3.8×10^{2}	.0038 seconds
2	10^{-6}	3.8×10^{4}	.38 seconds
3	10^{-7}	3.8×10^{6}	38 seconds
4	10^{-8}	3.8×10^{8}	1.05 hours
5	10^{-9}	3.8×10^{10}	4.4 days
6	10^{-10}	3.8×10^{12}	1.2 years

(b) Predicted Timings

Table 1
The number of iterations n required to achieve a degree of precision of the timer overhead up to 5 significant digits (SD) based on equation 5 and the given parameter settings. Also shown is an estimate of the time each experiment would take.

timing penalty. In the next section we discuss the tractability of the above technique and a solution toward reducing the time required to undertake a timing analysis of Java bytecode instructions.

2.3 Model tractability

In order for the above technique to be used, we require that the amount of time involved in performing the experiment be within reasonable bounds. Table 1(b) shows a number of possible experiment durations based on the constraints shown in Table 1(a). For example, Table 1(a) represents an experiment at the 95% confidence level, timing a code segment of length approximately 10^{-5} seconds using a clock with a resolution of 10^{-3} seconds. From Table 1(b) we can see, for example, to time a code segment to 4 significant digits would take approximately 1.05 hours, whereas to time a code segment to 6 significant digits would require approximately 1.2 years.

In the previous subsection we discussed the timing of Java bytecode using our technique. We note that the example given would only estimate the execution of the instruction to 1 significant digit. If we require any further precision for example, say 3 significant digits, then the experiment would take approximately 1.2 years.

From equation 5 we see that reducing the number of significant digits reduces the overall time required to run the experiments. By increasing the bytecode sequence to be timed by a factor of 10, the effect of which would reduce the overall execution time considerably. In our experiments we choose a sequence length of 1000 instructions, and estimate its duration to 4 significant digits.

2.4 Related work

There has been relatively little work documented in relation to Java bytecode instruction timings compared to other optimisation directed research. For example, a considerable body of work has been carried out primarily dealing with quantitative analysis as a way of generalizing possible optimisation opportunities for JVM performance. Daly et al. undertook a study of dynamic Java instruction frequencies for Java programs taken from the Java Grande benchmark suite and analyse local variable array size, operand stack size as well as quantifying the number of Java method parameters [5,8]. Gregg et al. perform a similar study of Java applications, although they concentrate on programs drawn from the SpecJVM98 benchmark suite [12,24]. Horgan et al. quantify the dissimilarity between programs taken from the Java Grande benchmark suite [15], while others focus on sequences of instructions known as n-grams [19,23]. Radhakrishnan et al. perform an analysis of instruction and data cache miss-rates within SpecJVM98 applications [21]. Dieckmann et al. present an empirical study of memory usage within SPECjvm98 applications [10]. Inoue et al. perform an analysis of the lifetime of objects within a number of Java programs [16].

In contrast, work on bytecode timing is relatively sparse. Herder et al. present the results for Java bytecode timing, although, the technique used to gather such results is not documented [13]. Wong et al. present a technique for the measurement of bytecode execution times, although, the final technique is not platform independent and relies on native method invocations [26].

The analysis of machine level instruction execution timings has also been undertaken. The work of Peuto et al. was directed toward the production of an instruction timing model to model CPU performance measurements [20]. Architecture manufacturers such as Intel and IBM also detail the execution time in clock ticks of the machine instructions associated with their architectures.

Albert et al. propose a framework for the cost analysis of Java bytecode [1]. This technique involves transforming the iterative bytecode structure to a recursive representation, and inferring cost relations from this representation. In [2], Albert et al. apply their previously proposed framework to Java programs that include operations such as: recursion, single loop methods, nested loops, list traversal as well as dynamic dispatching.

3 Experimental Design and Methodology

Stark et al. presented a decomposition of the JVM into a number of sub-machines [22]. Each sub-machine has the ability to execute particular subsets of the JVM instruction set. We also adopt this approach, by concentrating on individual cores of the JVM and their respective programming paradigms, and believe that a clearer understanding of the interaction of a Java application and JVM can be gained. In particular, we concentrate on reporting timing results of the 137 JVM instructions that compose the imperative core of the JVM.

The platform independence of our technique is insured by using only standard Java library timing methods. There are two timing methods available for consideration: *System.currentTimeMillis* and *System.nanoTime*. Our choice was based on timing precision and accuracy and as such the former of the two was chosen. This choice was made as *System.nanoTime* cannot guarantee nanosecond accuracy. All Java timing classes, containing the JVM byte code instruction to be timed, were engineered using the byte code engineering library (BCEL) [3].

The experimental procedure first involved estimating the duration of the timer overhead using the technique of Section 2.2. The timer overhead is the time required to perform two successive calls of the Java timing method *Sys-*

tem.currentTimeMillis.

All experiments conducted where carried out on a Rocks Linux cluster containing 100 nodes. Each node contained one 1.13GHz Intel Pentium III dual core processor with 1Gb of RAM and a cache size of 512Kb running the Linux CentOS operating system release 4.4 (kernel version 2.6.9) executing at run level 3 with no X-Server.

The Sun Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0.07-b03) was used to run all Java classes and the JVM was run in interpretor mode.

3.1 Bytecode Sequence Generation

The estimation of the time required to execute individual Java bytecode instructions involved inserting sequences of the instruction of interest between two timing method calls. For example, to estimate the time required to execute four *iconst_0* instructions, these instructions were placed between two calls of the timing method *System.currentTimeMillis* as shown in code segment 1.

One execution of the sequence of instructions depicted in code segment 1 represents one Bernoulli trial. To estimate the execution time of this sequence to a number of significant digits and a certain confidence level, requires that the sequence be executed n times within a loop. JVM operand stack constraints require that the operand stack height be the same upon entry and exit of a loop. As such it is required that a stack balancing be performed after the final call to the timer method as shown in code segment 1.

Code Segment 1 The insertion of four *iconst_0* instructions between two timing method invocations and relevant stack balancing instructions.

```
0:    invokestatic  #2; //Method System.currentTimeMillis:()J
3:    lstore_1
4:    iconst_0
5:    iconst_0
6:    iconst_0
7:    iconst_0
8:    invokestatic  #2; //Method System.currentTimeMillis:()J
11:   lstore_3
12:   pop
13:   pop
14:   pop
15:   pop
```

In some cases it was required that the Java local variable array be initialized with values of the appropriate type which where required by bytecodes within the instruction sequence to be timed. For example, code segment 2 shows a sequence of four *iload_0* instructions to timed. The Java instruction *iload_0* requires that the value stored in the local variable array indexed at position 0 be of type *int*. To accommodate this precondition and to ensure that the state of the Java method be consistent, local variable array types could be ensured by passing the appropriate type as an argument to the method. This technique also generalizes the timing method, and the effect of changing the value can be controlled by the caller.

Code Segment 2 An instruction sequence containing four *iload_0* instructions. The local variable array at index 0 is guaranteed to contain an *int* value due to the first method parameter type.

```
public static void timeILOAD_0(int);
  Code:
   0:   invokestatic  #2; //Method System.currentTimeMillis:()J
   3:   lstore_1
   4:   iload_0
   5:   iload_0
   6:   iload_0
   7:   iload_0
   8:   invokestatic  #2; //Method System.currentTimeMillis:()J
   11:  lstore  4
   12:  return
```

We also note that in some particular cases a continuous sequence of a particular instruction could not be generated. For example the *i2d* instruction requires one operand of type *int* to be on top of the operand stack and leaves a value of type *long* on the top of the operand stack. This value must be popped from the top of the stack in order for the next *i2d* instruction to execute. Code segment 3 shows the method bytecode for a sequence of two *i2d* instructions. This instruction sequence also depicts the scenario where an instruction requires a particular value to be present on top of the operand stack. In this case a value of type *int* is guaranteed by the initial *iconst_0* instruction sequence.

In the situation where the instruction of interest is immediately followed by a *pop* or *pop2* instruction, the time for this event would be adjusted by subtracting the duration of the stack manipulating instruction.

4 Platform-Independent Timing Results

In this section, we present our results. First we present the timing overhead associated with our technique, secondly we present the execution times for all imperative bytecode instructions, and finally we present the results of a cluster analysis performed on the bytecode instruction timings.

4.1 Timer Overhead

As previously discussed, initial experiments have identified that timer overhead significantly over-shadows the execution time of a single bytecode instruction. The timing of bytecode sequences and the adjustment of the recorded times to eliminate timer overhead requires that we estimate the timer overhead to a number of significant digits.

Code Segment 3 An instruction sequence containing two *i2d* instructions and the operand stack balancing instructions *pop2*. Also depicted is the inclusion of the instructions *iconst_0*, to insure that the JVM stack contains the appropriate instruction operands.

```
public static void timeI2D();
  Code:
   0:   iconst_0
   1:   iconst_0
   2:   invokestatic  #2; //Method System.currentTimeMillis:()J
   5:   lstore_0
   6:   i2d
   7:   pop2
   8:   i2d
   9:   pop2
   10:  invokestatic  #2; //Method System.currentTimeMillis:()J
   13:  lstore  3
   14:  return
```

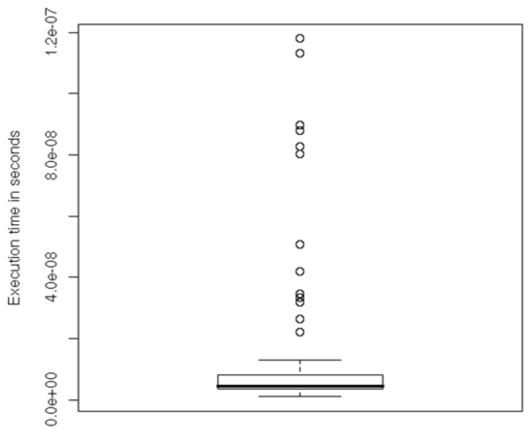
SD	Timer overhead (sec)
1	$\overbrace{0.01}u_2u_3u_4u_5u_6u_7u_8 \times 10^{-3}$
2	$\overbrace{0.013}u_3u_4u_5u_6u_7u_8 \times 10^{-3}$
3	$\overbrace{0.0131}u_4u_5u_6u_7u_8 \times 10^{-3}$
4	$\overbrace{0.01318}u_5u_6u_7u_8 \times 10^{-3}$
5	$\overbrace{0.013186}u_6u_7u_8 \times 10^{-3}$

Table 2
The timer overhead associated with two successive calls to the Java timer *System.currentTimeMillis* correct to *SD* significant digits.

Table 2 shows the magnitude of the timer overhead, estimated up-to five significant digits. Column, **SD**, represents the number of significant digits representing the timer overhead result shown in column **timer overhead**. For example, estimated to five significant digits the timer overhead is recorded as 1.3186×10^{-5} seconds.

4.2 Bytecode Execution Timings

Figure 2 summarises the timings for the 137 instructions using a box plot and the corresponding six point statistical summary of the instruction execution ranges in seconds. Note that in Figure 2 the *Min* and *Max* values represent the two whiskers in the box plot, and bound the instructions not falling within the outlier set which represent approximately 91% of the imperative instructions timed. 50% of those fall within the first and third quartiles (by definition), 24% fall between the minimum and first quartile while the remainder, 16% fall between the third quartile and the maximum. Those instructions falling between the first quartile and third quartile have execution times within a range of 4.488×10^{-9} seconds.



Min.	1st Qu.	Median	3rd Qu.	Max.	Mean
1.093×10^{-9}	3.583×10^{-9}	4.527×10^{-9}	8.072×10^{-9}	1.299×10^{-8}	1.061×10^{-9}

Fig. 2. A box plot and the six point summary for the 137 JVM instruction timing results measured using a low resolution timer. The six point summary gives the minimum and maximum values for the whiskers as well as the first and third quartiles, and the median and mean of the data set. All measurements are in seconds.

From Figure 2 we can identify 13 instructions whose execution times deviate significantly from the execution times of the remaining instructions. These outliers are listed in Table 3 and can be characterised as falling into three broad categories.

The first category in the outlier set represents instructions that perform a primitive type conversion. In particular, we can see that conversion from a floating-point representation to either a *long* or an *int* value, and vice versa, takes a considerable amount of time. Conversions from different number representations take approximately 10 times as long as conversions from floating-point to floating-point and integer to integer.

The second category of instructions forming the outlier set, represent instructions that perform a remainder operation. For example, *irem*, *lrem*, *frem* and *drem*. These instructions are found within class files whose Java source code would include modulo operations. These instructions take in general, 10 times longer to execute compared to the other imperative instructions.

The third category identified within the outlier set, are the *ldiv* and *idiv* instructions. The outlier set would seem to suggest that computationally intensive programs would take significantly longer to execute compared to other imperative programs.

With the aim of characterising instruction execution times further, we have performed a cluster analysis of the instruction execution times. We present the results of the cluster analysis next.

Inst	Time (sec)
lrem	1.180681×10^{-7}
ldiv	1.132759×10^{-7}
d2i	8.976430×10^{-8}
d2l	8.795759×10^{-8}

Inst	Time (sec)
f2l	8.267345×10^{-8}
f2i	8.028337×10^{-8}
drem	5.049472×10^{-8}
frem	4.191904×10^{-8}
idiv	3.449739×10^{-8}

Inst	Time (sec)
l2d	3.321960×10^{-8}
irem	3.179264×10^{-8}
i2d	2.635641×10^{-8}
l2f	2.205176×10^{-8}

Table 3
The 13 instructions in the outlier set along with their corresponding execution times in seconds.

	Granularity (sec)	Number of Groups
Min:	1.093×10^{-9}	24
1st Qu:	3.584×10^{-9}	14
Median:	4.527×10^{-9}	11
3rd Qu:	8.072×10^{-9}	9
Mean:	1.061×10^{-8}	7
Max:	1.181×10^{-7}	1

Table 4
The number of cluster groups resulting from using six different granularities for the cluster distance metric.

4.3 Instruction Cluster Analysis

The identification of a number of groups of instructions, where each instruction within a group exhibits similar execution time, would help in reducing the dimensionality of any timing model dependent on knowing the execution time of each JVM instruction. In performing the cluster analysis we choose a *granularity value* as a cut-off point, so that when the instructions in a cluster differ by less than the granularity value we do not subdivide them further. Thus there is a trade-off: small granularity values give better accuracy at the cost of a larger number of groupings.

As an example, we consider the possibility of setting the granularity value based on six statistics: the minimum value, the 1st quartile, the median, the 3rd quartile, the mean and the maximum value of all instruction durations. Table 4 shows how the total number of clusters varies as we change the granularity. The first column of Table 4 shows the relevant granularity and column two shows the number of groupings for that level of granularity. For example, clustering instructions based on the minimum instruction execution time produces 24 groups of instructions. By definition, using the maximum value will always group all instructions into one cluster.

The choice of a specific granularity will ultimately depend on the computational task at hand. However, as an example, Table 5 shows the result of clustering all 137 instruction times based on the median value, producing 11 groups. In Table 6 we show the instructions that form each group. The groups are listed in increasing order of instruction duration, with the instructions in group (a) executing the fastest and those in group (k) having the slowest execution time.

In Table 7 we present three box-plots for the largest cluster groups (a, b and c), which show the spread of the instructions times within each group. The instructions in cluster groups (d) through (k) are all outliers whose timings have already been presented in Table 3. Approximately 91% of all the instructions fall within the

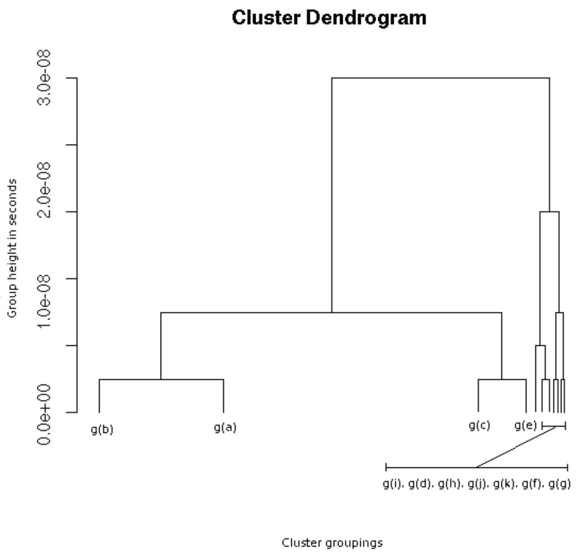


Table 5
A dendrogram showing 11 clusters of groups of instructions. These clusters are based on the instruction timings where we ignore differences of less than the median of the whose data set.

Group	Instructions in this group
(a)	aconst_null, iconst_m1, iconst_[0-5], lconst_[0-1], fconst_[0-1], bipush, sipush, iload_[n-3], fload_[n-3], lload_[0-3], fstore_[n-3], istore_[0-3], land, lor, lxor, iinc, d2f, fcmpl, dcmpg, goto, ladd, isub, lsub, imul, ineg, lneg, fneg, dneg, iand, ior, ixor, i2l, l2i, i2b, i2c, i2s, lcmp, fcmpg, dcmpl, ifeq, ifne, iflt, ifge, ifgt, ifle, ifieq, ifine, ifilt ifigt
(b)	nop, fconst_2, dconst_0, dconst_1, lload_n, dload_[n-3], istore_n, lstore_n, dstore_n, lstore_[0-3], dstore_[1-2] dupx1, dupx2, dup2x1, fadd, fsub, lmul, ishl, ishr, iushr dstore_0, dstore_3, pop, pop2, dup, dup2, swap, iadd
(c)	dup2x2, dadd, dsub, fmul, dmul, fdiv, ddiv, lshl, lshr, lushr, i2f, f2d, ifige, ifile
(d)	i2d, l2f
(e)	idiv, irem, l2d
(f)	frem
(g)	drem
(h)	f2i, f2l
(i)	d2i, d2l
(j)	ldiv
(k)	lrem

Table 6
The instructions in each of the 11 cluster groups obtained using the median as the minimum granularity for the cluster distance metric. The 11 groups are listed in order of increasing instruction time.

first three groups. As previously identified in Table 3, floating point operations take considerably longer to execute compared to non integer instructions and this coincides with our intuition that floating point operations take considerably longer than ALU (arithmetic logic unit) bound instructions.

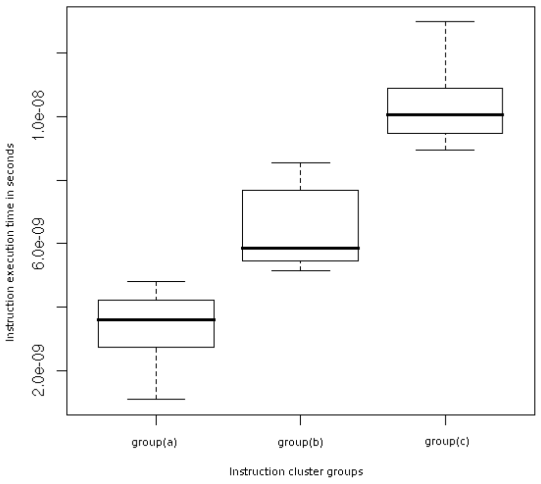


Table 7
A box-plot showing the distribution of instruction times within the three largest cluster groups (a), (b) and (c). These three groups account for 124 of the 137 instructions studied.

5 Platform independent timing vs RDTSC

In this section, we calibrate all instruction execution times recorded using our platform independent timing technique against a set of instruction timings gathered using the RDTSC (read time stamp counter) instruction of the Intel Pentium processor [27].

The technique for capturing the duration of an instruction in clock ticks requires that a specific register be accessed through an RDTSC assembly instruction call. As Java does not directly support assembly code within class files, the timing method had to be written in C and linked to the JVM on start-up. The registering of the native timing method was done through the Java Virtual Machine Profiler Interface (JVMPI) [17]. Using this technique the 137 imperative instructions were timed. For comparison with the platform independent timing strategy, we also timed instruction sequences containing 1000 instructions using the RDTSC assembly instruction.

5.1 Linear correlation

In order to compare the results achieved using our platform independent technique with the results using the RDTSC instruction we measured the linear correlation using the Pearson correlation coefficient statistic. Equation 7 shows the Pearson empirical correlation coefficient statistic. The variable x_i ranges over the instruction times for each of the 137 instructions analysed using our platform independent measure, $\overline{x}$ represents the mean value of this data set. Similarly y_i and $\overline{y}$ represent the individual instruction times and their associated mean value for the times

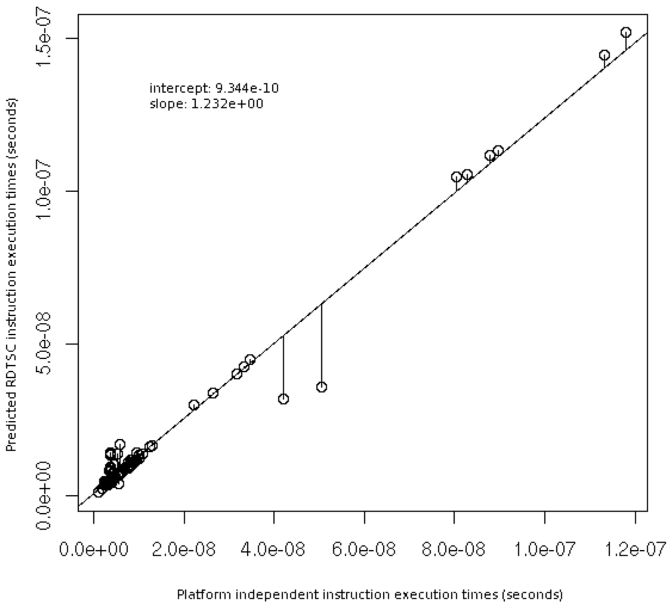


Table 8
A scatter-plot comparing the platform-independent and RDTSC instruction timings for each of the 137 instructions. The line predicted by the linear regression model is also shown.

recorded using the RDTSC assembly instruction.

$$r = \frac{\sum(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum(x_i - \overline{x})^2 \sum(y_i - \overline{y})^2}} \tag{7}$$

The Pearson correlation coefficient is a symmetric measure of the association between two variables. It ranges from −1 to 1, where the extreme values indicate a perfect correlation and 0 means no correlation. The result of this analysis is that r is equal to 0.9887898, which indicates a very strong positive linear correlation between the two data sets. In order to model this linear relationship we undertook a linear regression analysis on both data sets.

5.2 Linear regression model

The linear regression model attempts to find the line that passes through the data points so as to minimize the distance from the point (x_i, y_i) to the fitted line. The regression model selects a value for α and β from equation 8 so that the sum of the squares is minimized.

$$y_i = \alpha + \beta x_i \tag{8}$$

The parameters α and β are given in Table 8 as the *intercept* and *slope* respectively. From these results we can see that the line of best fit intercepts the y-axis at 9.344×10^{-10} and has a slope of 1.232. Thus according to this linear model our platform-independent measures under-predict the instruction execution times recorded using the RDTSC method by approximately 23%. While this difference merits further study, the strength of the linear relationship at least allows a once-off platform-dependent calibration to be extended over all the instructions. The two values with the largest residuals represent the *frem* and *drem* instructions, with the linear model over-predicting these two instructions by 2.181×10^{-8} seconds and 1.6703×10^{-8} seconds respectively.

6 Conclusion and Future Work

In this paper we have presented a technique for the timing of Java bytecode instructions that is platform independent. We have investigated the effect of timer overhead and the importance of subtracting this quantity from instruction timings. We have characterised the execution times of all Java imperative instructions. We have considered the clustering of instructions based on their execution times and finally we have presented a comparison of our technique against instruction timings acquired using the RDTSC assembly instruction.

The contributions of this paper are: First we have presented a technique that statistically estimates the execution time of Java instructions within a particular confidence level. In particular, we can quantify the error associated with each instruction timing. We have characterised instruction execution times and have identified a group of imperative instructions, primarily floating point conversion instructions, that execute considerably slower than all other instructions. We have presented a technique that clusters instruction timings within a predefined granularity. Finally we have identified a strong positive linear relationship between instruction times acquired using our statistical method and those acquired using the RDTSC assembly instruction. We have modeled this linear relationship and have identified that platform independent instruction timing analysis under estimates the execution times of instructions by approximately 23%. However, the strength of the linear model still allows us to accurately calibrate the measurements.

For future work, we intend to quantify the effect of processor pipelines and cache miss rates on instruction timings. We also intend to carry out our experiments on different platforms and investigate the correlation of results acquired from different JVM implementations. We also intend to extend our instruction timing model to include instructions from within other cores of the JVM. As part of future work we also intend on applying our results to existing JVM models. In particular, we intend on using our instruction execution times, as part of a larger model to predict the execution times of Java applications.

References

[1] Albert, E., P. Arenas, S. Genaim, G. Puebla and D. Zanardini, *Cost analysis of Java bytecode*, in: *European Symposium on Programming*, Braga, Portugal, 2007, pp. 157–172.

[2] Albert, E., P. Arenas, S. Genaim, G. Puebla and D. Zanardini, *Experiments in cost analysis of Java bytecode*, Electron. Notes Theor. Comput. Sci. **190** (2007), pp. 67–83.

[3] Apache Software Foundation, *BCEL: The bytecode engineering library* (2006).
URL http://jakarta.apache.org/bcel/

[4] Beilner, H., *Measuring with slow clocks*, Technical Report 88-003, International Computer Science Institute, Berkeley, California (1988).

[5] Bull, M., L. Smith, M. Westhead, D. Henty and R. Davey, *Benchmarking Java Grande applications*, in: *Second International Conference and Exhibition on the Practical Application of Java*, Manchester, UK, 2000, pp. 63–73.

[6] Burke, M., J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Sreedhar, H. Strinivasan and J. Whaley, *The Jalapeno dynamic optimising compiler for Java*, in: *Java Grande Conference*, San Francisco, CA, USA, 1999, pp. 129–141.

[7] Cierniak, M., M. Eng, N. Glew, B. Lewis and J. Stichnoth, *The open runtime platform: A flexible high-performance managed runtime environment*, Intel Technology Journal **7** (2003), pp. 5–18.

[8] Daly, C., J. Horgan, J. F. Power and J. T. Waldron, *Platform Independent Dynamic Java Virtual Machine Analysis: the Java Grande Forum Benchmark Suite*, in: *Joint ACM Java Grande - ISCOPE Conference*, Stanford, CA, USA, 2001, pp. 106–115.

[9] Danzig, P. B. and S. Melvin, *High resolution timing with low resolution clocks and microsecond resolution timer for sun workstations*, SIGOPS Oper. Syst. Rev. **24** (1990), pp. 23–26.

[10] Dieckmann, S. and U. Hölzle, *A study of the allocation behaviour of the SPECjvm98 Java benchmarks*, in: *13th European Conference on Object-Oriented Programming*, Lisbon, Portugal, 1999, pp. 92–115.

[11] Gagnon, E., “A Portable Research Framework for the Execution of Java Bytecode,” Ph.D. thesis, McGill University (2002).

[12] Gregg, D., J. 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*, Kluwer Academic, 2002 pp. 1–18.

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

[14] Hogg, R. and A. Craig, “Introduction to Mathematical Statistics,” Prentice Hall, 1995.

[15] Horgan, J., J. F. Power and J. T. Waldron, *Measurement and Analysis of Runtime Profiling Data for Java Programs*, in: *First IEEE International Workshop on Source Code Analysis and Manipulation*, Florence, Italy, 2001, pp. 122–130.

[16] Inoue, H., D. Stefanovic and S. Forrest, *On the prediction of Java object lifetimes*, IEEE Transactions on Computers **55** (2006), pp. 880–992.

[17] JVMPI, *Java Virtual Machine Profiler Interface*, Sun Microsystems, Inc.
URL http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/

[18] Lilja, D. J., “Measuring Computer Performance: A practitioner’s guide,” Cambridge University Press, 2000.

[19] O’Donoghue, D., A. Leddy, J. 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*, Dublin, Ireland, 2002, pp. 187–192.

[20] Peuto, B. L. and L. J. Shustek, *An instruction timing model of cpu performance*, in: *Proceedings of the 4th Annual Symposium on Computer Architecture*, 1977, pp. 165–178.

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

[22] Stark, R., J. Schmid and E. Borger, “Java and the Java Virtual Machine Definition, Verification, Validation,” Springer, 2001.

[23] Stephenson, B. and W. Holst, *A quantitative analysis of Java bytecode sequences*, in: *3rd International Symposium on Principles and Practice of Programming in Java*, 2004, pp. 15–20.

[24] The Standard Performance Evaluation Corporation, *SPEC releases SPEC JVM98 Benchmarks, First industry-standard benchmark for measuring Java performance.*, Press release (1998).
URL http://www.specbench.org/osg/jvm98/press.html

[25] Wilkinson, T., *Kaffe: A free virtual machine to run Java code.* (1997).
URL http://www.kaffe.org/

[26] Wong, P., *Bytecode monitoring of Java programs*, BSc project report, University of Warwick, UK (2003).

[27] Work, P. and K. Nguyen, *Measure code sections using the enhanced timer*, Intel Corporation (2006). URL http://www.intel.com/cd/ids/developer/asmo-na/eng/209859.htm?page=1