

Bytecode Monitoring of Java Programs

Author: Peter Wong

Supervisor: Stephen Jarvis

Content

Abstract	5
Keywords	5
Chapter 1 Background and Motivation	6
1.1 PACE – Performance Analysis and Characterisation Environment	6
1.2 PACE Toolset Components	9
1.3 Model Characterisation Separation	10
1.4 The Layered framework	11
1.5 Java World – an extension to PACE	13
Chapter 2 Project Objectives and Specification	14
2.1 Methodologies	14
2.2 Java’s Virtual Machine, Instruction Set and Assembler	15
2.3 XML Characterisation	19
2.4 Main programming languages in used	20
Chapter 3 Design and Implementation (1st edition)	22
3.1 Finding methods to calculate running times of bytecodes	22
3.2 From Template model to implementation	27
3.3 Component’s details	27
3.4 Mechanics of the Bytecode Prediction Template	31
3.5 Toolkit Development	35
3.6 Formal Evaluation	42
3.7 Interpretation of the measured timings of bytecodes	42
3.8 Observation	49

Chapter 4	Development	50
4.1	Initial Thoughts and Experiments	50
4.2	Bytecode latency from invoking native method	50
4.3	The latency in variable assignment and retrieval	51
4.4	The effects of Hot Spot compiler and Java Optimisation.	53
4.5	Hot Spot motivation	54
4.6	Hot Spot detection	56
4.7	Dynamic de-optimisation	57
4.8	Developed Ideas	58
4.9	From ideas to Implementation	60
Chapter 5	Design and Implementation (2nd edition)	66
5.1	Bytecode Block Definition (Sequential Bytecode Block)	66
5.2	Benchmarking Bytecode Blocks	70
5.3	Implementation of benchmark toolkit (revised edition)	70
5.4	A Theoretical Hypothesis	70
5.5	Components of benchmark toolkit	71
5.6	Preliminary Results and Understanding	77
5.7	High Performance Application (Benchmarks)	81
5.8	Further refining the evaluation bytecode sequence	82
5.9	Negative Valuation	86

Chapter 6	Result Evaluation	88
6.1	Fast Fourier Transform Benchmarks	90
6.2	Excessively Parallel Benchmarks	96
6.3	Other Benchmarks	100
Chapter 7	Conclusion	101
7.1	Summary	101
7.2	Limitation	105
7.3	Future Direction	106
Reference		108
Appendix		On DISK

Abstract

A performance prediction system (PACE – Performance Analysis Characterisation Environment) has been implemented to characterise the performance of C, Fortran and Mathematica codes. With the current increase in the popularity of the Java platform, PACE is being extended to characterise and predict distributed Java applications within dynamic heterogeneous environments. With the modern implementations of the Java Virtual Machine being able to carry out on-the-fly optimisations, Java methods are to be characterised as a control flow of bytecode blocks, rather than individual bytecodes. These bytecode blocks are then benchmarked to create a bank of predictive data for evaluating performance critical Java applications. This report describes the implementation of defining and monitoring these bytecode blocks and also evaluates the techniques that have been used.

Keywords: Java Virtual Machine
PACE
Hot Spot
Java Optimisation
Sequential Bytecode Blocks
Bytecode Prediction Template

Chapter 1

Background and Motivation

This chapter introduces the core idea of performance analysis for high performance computation on a Grid computing environment. It illustrates the framework of performance characterisation and how it leads to have the need to implement a Java bytecode monitor.

The computing architectural landscape is changing. Resource pools that were once large, multi-processor supercomputing systems are being increasingly replaced by heterogeneous commodity PCs and complex powerful servers. These new architectural solutions, including the Internet computing model [10] and the grid computing [11, 12] paradigm, aim to create integrated computational and collaborative environments that provide technology and infrastructure support for the efficient use of remote high-end computing platforms. The notion of so-called grid computing or the use of a **computational grid** is applying the resources of many computers in a network to a single problem at the same time - usually to a scientific or technical problem that requires a great number of computer processing cycles or access to large amounts of data. A well-known example of grid computing in the public domain is the ongoing SETI (Search for Extraterrestrial Intelligence) @Home project in which thousands of people are sharing the unused processor cycles of their PCs in the vast search for signs of "rational" signals from outer space. According to John Patrick, IBM's vice-president for Internet strategies, "the next big thing will be grid computing."

The success of these architectures relies on the outcome of a number of important research areas; one of these – **performance** – is fundamental, as the uptake of these approaches relies on their ability to provide a steady and reliable source of capacity and capability computing power, particularly if they are to become the computing platforms of choice.

The study of performance in relation to computer hardware and software has been a topic of much scrutiny for a number of years. It is likely that this topic will change to reflect the emergence of geographically dispersed networks of computing resources such as

grids. There will be an increased need for high performance resource allocation services and an additional requirement for increased system adaptability in order to respond to the variations in user demands and resource availability. Performance engineering in this context raises a number of important questions and one question of which the motivation of this project is based on. Its answer will impact on the utilisation and effectiveness of related performance services:

How is this performance data obtained?

Gathering performance data can be achieved by number of methods. Monitoring services provide records (libraries) of dynamic information such as resource usage or characteristics of application execution. This data can be used as a benchmark for anticipating the future performance behaviour of an application, a technique that can be used to extrapolate a wide range of predictive results [13]. Alternatively it is possible to extract data from an application through the evaluation of analytical models. While these have the advantage of deriving *a priori* performance data – the application need not be run before performance data can be collected – they are offset by the complexity of model generation.

For the last 10 years the High Performance Systems Group has made significant contribution towards this field of research, namely a unique characterisation environment implemented with a toolkit PACE (Performance Analysis and Characterization Environment).

1.1 PACE - Performance Analysis and Characterization Environment

Performance Analysis and Characterization Environment (PACE) [7] provides a framework for developers to create detailed analytical performance models that can be used to predict the performance of their applications. It has been verified by the UK Defence Electronic Research Agency (DERA) that a predictive accuracy of less than 10% can be achieved using this technique. The system works by characterizing the application and the underlying hardware on which the application is to be run, and combining the

resulting models to derive predictive execution data. PACE provides the capability for the rapid calculation of performance estimates without sacrificing performance accuracy. PACE also offers a mechanism for evaluating performance scenarios – for example the scaling effect of increasing the number of processors – and the impact of modifying the mapping strategies (of process to processor) and underlying computational algorithms [9].

Details of the PACE toolkit can be seen in Figure 1.

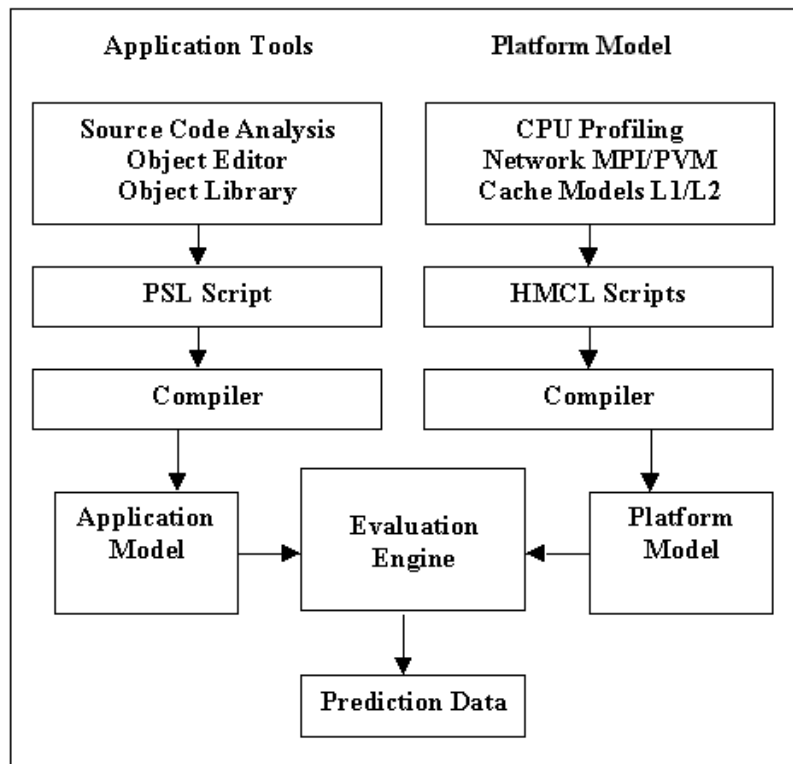


Figure 1 An outline of the PACE system including the application and platform (resource) modelling components and the parametric evaluation engine which combines the two

1.2 PACE Toolset Components

The PACE toolset includes a range of components that assists a user to create models, visualize results, use pre-defined models from a library, and use information derived from existing application codes. The number of vital components of PACE are described briefly below. [6]

- ***Evaluation Engine*** evaluates the current performance model, producing predictions of time, scaling, and resource usage.
- ***Workbench*** provides a user-friendly interface to the components of PACE.
- ***Source Code Analyzer*** assists the user in converting sequential source code into the CHIP³S performance language. The user directs this operation by specifying which code are associated with which sub-task elements. Currently this component enables C source to be input, using both parsing and profiling information.
- ***Object browser*** assists the user to scan predefined model libraries of application kernels, parallelisation strategies (parallel templates), and hardware models. The user may also define new library models.
- ***Object Editor*** assists the user to enter and review individual objects contained within the performance model.
- ***Parametric visualization*** enables application and/or system parameters to be varied, and provides a means in which the results can be visualized. Currently supports single and dual parameter manipulation.
- ***Trace visualization*** enables the visualization of a single prediction scenario. It provides time-space diagrams illustrating computation, communication and idle stages of processors. Currently, this analysis is provided by a trace data file link to the ParaGraph parallel monitoring system.

1.3 Model Characterisation Separation

An important feature of this design is that the separation of application and platform models and there are independent tools for each.

- **Application Tools** provide a means of capturing the performance aspects of an application and its “*parallelisation*” strategy. Static source code analysis forms the basis of this process, drawing on the control flow of the application, the frequency at which operations are performed, and the communication structure. The resulting performance specification language (PSL) scripts can be compiled to an application model. Although a large part of this process is automated, users can modify the performance scripts to account for data-dependent parameters and also utilise previously generated scripts stored in an object library.

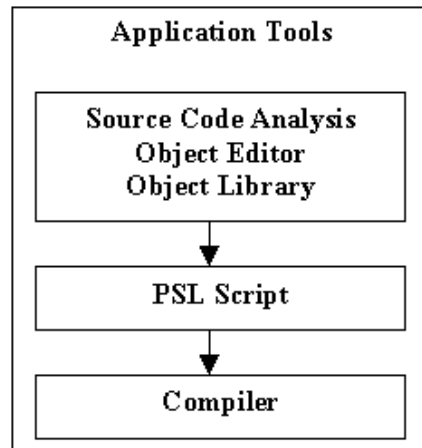


Figure 2 Application Model

- **Platform (Resource) Tools** model the capabilities of the available computing resources. These tools use a hardware modelling and configuration language (HMCL) to define the performance of the underlying hardware. The platform tools contain a number of benchmarking programs that allow the performance of the CPU, network and memory components of a variety of hardware platforms to be measured. The HMCL scripts provide a resource model for each hardware component in the system, since these models are (currently) static, once a model has been created for a particular hardware, it can be archived and reused.

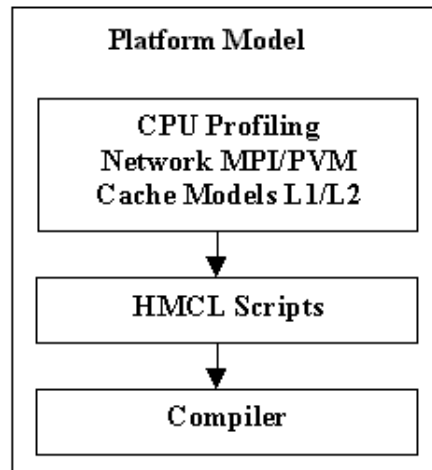


Figure 3 Platform (Resource) Model

1.4 The Layered framework

These models are created by describing an application's performance in a characterization language called CHIP³S [9]. CHIP³S encompasses a layered framework for performance characterization, as seen in Figure 4(a) [5]. Each layer can contain a number of objects that describe specific performance-critical elements of an application: subtask objects describe sequential elements of an application; parallel templates describe the parallelisation strategy of, and communication between, these subtasks; hardware objects characterize the computational and inter-communication performance of hardware resources. This inherent separation between hardware and software components allows predictions of the same application on different hardware platforms a case of simply inter-changing the model's hardware object as appropriate.

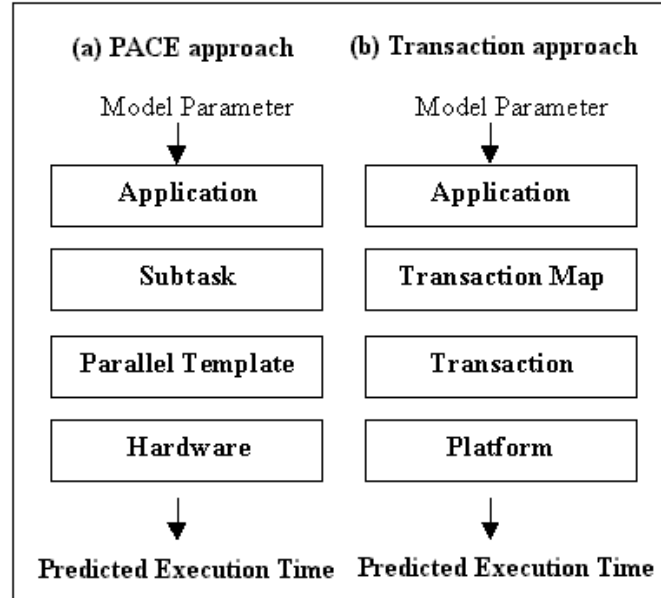


Figure 4 The Layered framework for performance characterisation: (a) The original PACE approach: (b) the revised transaction-based approach

PACE provides tools to characterize the performance of C, Fortran and Mathematica codes. Once the application and hardware models have been built, they can be evaluated using the PACE *Evaluation Engine*. PACE allows: time predictions (for different systems, mapping strategies and algorithms) to be evaluated; the scalability of the application and resources to be explored; system resource usage to be predicted (network usage, computation, idle time etc), and predictive traces to be generated through the use of standard visualisation tools.

Such subtasks characterizations within CHIP³S is achieved by using a tool called ‘capp’ that processes methods defined within C source code and outputs performance characterizations of these methods in the CHIP³S language. These characterizations are a parameterized control flow of a number of atomic instructions that map onto a set of common machine instructions. Another tool benchmarks these machine instructions for a given resource, providing a list of timings that are included within the resource’s associated hardware object. Evaluating a subtask to predict its performance involves essentially adding up all the timings for the machine instructions that are to be executed for a given application run.

1.5 Java World – an extension to PACE

With the current increase in the popularity of the Java platform, as well as the interest in computational Grids within the high performance community, PACE is being extended to characterize and predict distributed Java applications within dynamic heterogeneous environments – known as **JPACE**. A new XML-based language [8] hence is being developed that uses a more flexible transaction-based approach to performance characterization; shown in Figure 4(ii). Applications are characterized as a number of transactions, or items of work, where their relation to each other is described within a transaction map.

Taking the original method of performance characterization, it would seem to be equivalent, where characterizing Java methods is concerned, to create a control flow of Java bytecodes. Each bytecode could be benchmarked in the same way as machine instructions are in CHIP³S, the culmination of which would result in the prediction of a Java method's performance, even though later on it has been discovered that due to on-the-fly optimizations within modern implementations of the JVM that this is not the case.

Figure 5 depicts a graphical description of the relationship between the benchmarked bytecode timings and the PACE system.

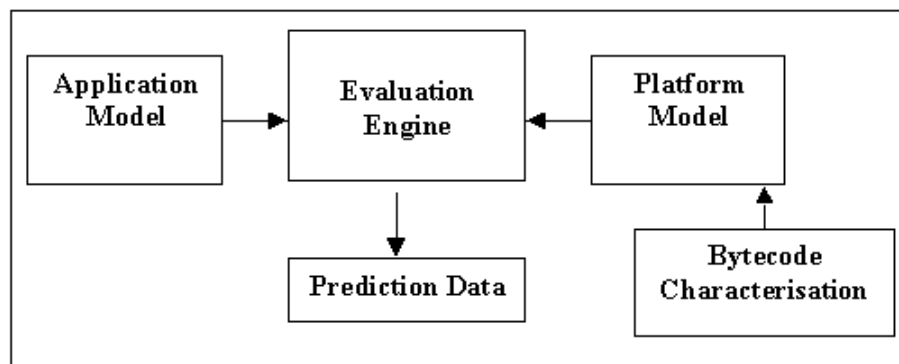


Figure 5 Structure of performance evaluation process

Chapter 2

Project's Objectives and Specifications

This chapter details the project's objectives and its specifications. It establishes the methodologies and hypotheses of which this project is set about to implement and experiment. It also identifies the main programming tools that are utilised throughout this project.

2.1 Methodologies

By having this motivation as the milestone in the performance study, this project has then adapted this initiative and with early background research on the current grounding, two possible benchmark methodologies were specified for this project:

- **Timing analysis of Java bytecodes** - An initiative that is brought up to investigate the implementation of benchmarking the **Java Virtual Machine (JVM)** Instruction Set, using the Java Assembler Interface called "Jasmin". It takes ASCII descriptions for Java classes, written in an assembler-like syntax and using the JVM instruction set. It converts them into binary Java class files suitable for loading into a JVM implementation. The initial idea is to benchmark single bytecode at a time by repetitively executing individual bytecode in multiples of 10s, 100s and 1000s, to enable JVM to monitor these bytecodes a technique so-called **Application Response Measurement (ARM)** [8] will be used to carry out timing analysis on that repetition. Further work could also be implemented to archive these timings across different architecture so that a readily available library of metric can be utilized to carry out performance prediction on Java programs.
- **Method prediction on Java Programs** - A initiative that is brought up to investigate the implementation of predicting a Java program performance by monitoring the bytecode activity on a method level of the program source code. This initiative will utilize the ARM by running several unique Java method at source code level and timing these method using ARM "method call" transaction method and using a Java Parser, these method code can be then analysed at a bytecode level. This method can be looked at as a form of generation of simultaneous equations where by different

types of bytecodes will represent mathematical variables with the occurrences as its multiples. Once enough methods are analysed, these equations will be able to solve and timing of bytecode can then be looked at and further be used to predict future Java programs.

2.2 Java's Virtual Machine, Instruction Set and Assembler

To understand the notion of utilising bytecode as a medium to benchmark Java applications running within some dynamic heterogeneous environments, it is necessary understand the details of this intermediary language.

The *Java Virtual Machine* (JVM) [2] is a platform-neutral runtime engine used to execute Java programs. During the execution of a Java program, the constituent instructions are not executed directly by the hardware provided by the architecture, instead an intermediary stage of bytecode interpretation is carried out by the Virtual Machine.

JVM could be viewed as a “virtual” processor and hence machine instructions had been implemented for this engine. The JVM instruction set is relatively similar to a set for a real CPU. A Java virtual machine instruction therefore consists of an opcode specifying the operation to be performed, followed by zero or more operands embodying values to be operated upon.

To utilise these bytecodes at a higher level, a tool is needed for constructing class files from textual description. For simplicity Jasmin is chosen, Jasmin is a Java Assembler. It takes ASCII descriptions for Java classes, written in a simple assembler-like syntax using the Java Virtual Machine instructions set. It converts them into binary Java class files suitable for loading into a Java interpreter.

Incidentally it should be noted that to execute Java methods, the execution engine retrieve and processes the corresponding bytecodes. Bytecode consists of a sequence of single byte opcodes, each of which identifies a specific operation to be carried out.

e.g. the opcode 96 represents the instruction `iadd`, which adds two integers.

Listing 1 and 2 give a brief description of instruction syntax used in Jasmin:

- ***Local variable instructions***

These instructions use local variables:

e.g.

```
aload 1    ; push local variable 1 onto the stack
ret 2      ; return to the address held in local variable 2
```

- ***The bipush, sipush and iinc instructions***

The bipush and sipush instructions take an integer as a parameter:

```
bipush <int>
sipush <int>
```

The iinc instruction takes two integer parameters:

```
iinc <var-num> <amount>
```

- ***Branch instructions***

These instructions take a label as a parameter:

e.g.

```
Label1:
    goto Label1    ; jump to the code at Label1
                   ; (an infinite loop!)
```

- ***Class and object operations***

These instructions take a class name as a parameter:

e.g.

```
new java/lang/String    ; create a new String object
```

- ***Method invocation***

These instructions are used to invoke methods:

e.g.

```
; invokes java.io.PrintStream.println(String);
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
```

- ***Field manipulation instructions***

The four instructions getfield, getstatic, putfield and putstatic have the form:

e.g.

```
; get java.lang.System.out, which is a PrintStream
getstatic java/lang/System/out Ljava/io/PrintStream;
```

Listing 1 a brief description of instruction syntax used in Jasmin

- ***The newarray instruction***

The newarray instruction is followed by the type of the array,
newarray <array-type>

e.g.

```
newarray int
newarray short
newarray float
etc.
```

- ***The multianewarray instruction***

The multianewarray instruction takes two parameters, the type descriptor for the array and the number of dimensions to allocate:

```
multianewarray <array-descriptor> <num-dimensions>
```

e.g.

```
multianewarray [[[I 2
```

- ***The ldc and ldc_w instructions***

The ldc and ldc_w instructions are followed by a constant:

```
ldc <constant>
ldc_w <constant>
```

- ***The lookupswitch instruction***

The lookupswitch instruction has the syntax:

```
<lookupswitch> ::=
lookupswitch
    <int1> : <label1>
    <int2> : <label2>
    ...
    default : <default-label>
```

- ***The tableswitch instruction***

The tableswitch instruction has the syntax:

```
<tableswitch> ::=
tableswitch <low>
    <label1>
    <label2>
    ...
    default : <default-label>
```

- ***No parameter***

These instructions (the majority) take no parameters:

e.g.

```
pop          ; remove the top item from the stack
iconst_1     ; push 1 onto the stack
swap         ; swap the top two items on the stack
```

Listing 2 a brief description of instruction syntax used in Jasmin (continued)

Other research work on Java bytecode analysis has also been considered. Such as a technical written by C. Herder and J. J. Dujmovic at San Francisco State University titled *Frequency Analysis and Timing of Java Bytecodes* [4] has been studied. To understand the characterization of Java workloads, bytecode execution times were measured. Their measured result were based on an Ultra Sparc workstation and SUN JDK 1.2.2. and since the testing machine architecture is similar to the machine that this project is based on, both results and implementation were very applicable towards this project.

As library of Java bytecodes is to be benchmarked, there needs some information to bind the benchmark results with the application being benchmarked. The way that applications are characterised in Application Response Measurement (ARM) would provide the relevant information.

2.3 XML Characterisation

As JPACE being implemented, a new XML-based language has been developed to cater the transaction approach of characterisation to which JPACE has adopted for a more dynamic characterisation.

This XML-based Transaction Definition Language (TDL) [8] is defined, which allows Java applications' performance critical component to be semantically defined, is an integral part of the technique for automatically ARming Java applications in accordance with the ARM 3.0 standard for Java. An application is instrumented with ARM method calls through a bytecode transformer prior to execution, providing ARM compliance while removing the necessity to modify (or even possess) the original Java source code.

Figure 6 shows an example of a TDL XML file defining two transactions of type method source and line number for the jar file `example2.jar`. The first transaction has the user name `admin` associated with it, and defaults to failing if the method `example2method` contained within the `example2class` class throws an exception. Two metrics are also associated with the transaction.

```

<?xml version="1.0"?>
<!DOCTYPE tdl SYSTEM "tdl.dtd">

<tdl jarfile="example2.jar">

  <transaction type="method_source" user_name="admin">
    <location class="example2class" method="example2method"/>
    <metric type="GuageFloat32" name="example2metric1" value="4.1"/>
    <metric type="String8" name="example2metric2" value="Load"/>
  </transaction>

  <transaction type="line_number" fail_on_exception="no" user_name="root">
    <location class="example2class2" method="example2method2"/>
    <line_number begin="32" end="208"/>
    <metric type="String8" name="example2metric3" value="MemUsage"/>
  </transaction>

</tdl>

```

Figure 6 An example of a TDL XML file

Hence, the XML characterisation files of Java applications would be utilised as a template towards effective analysis of micro-benchmark timings.

Now having decided the medium of which the performance benchmark will be based in, a set of programming languages for implementing benchmarking toolkit is to be chosen.

2.4 Main programming languages in used

Java(TM) 2 Runtime Environment (Java 2 SDK 1.4.1)

This version of the virtual machine is installed and readily available at the Department of Computer Science.

PERL – **P**ractical **E**xtraction and **R**eport Language (v5.0, v5.6.1)

This is the preferred script language for implementing toolkits mainly because by utilising functionalities heavily from C, *sed*, *awk*, and the Unix shells, Perl has become the language of choice for many I/O, file processing and management, process management, and system administration tasks. Since the process of bytecode monitoring requires certain amounts of ASCII files manipulation. [3]

Jasmin – **J**ava **A**Sse**M**bler **I**Nterface (v1.05)

Jasmin is a free Java assembler provided on the Internet. This is a tool for constructing class files from textual descriptions. These textual descriptions from classes are written in Java Virtual Machine instruction set which are converted into binary class files. Nevertheless, similarly to Jasmin there are also other Java assemblers available for free distribution, one such freeware is an assembler called Oolong and it was created with a counterpart i.e. a disassembler called Gnoloo, which in the course of this project became a very useful tool. [1]

JNI - **J**ava **N**ative **I**nterface

The Java Native Interface (JNI) is the native programming interface for Java that is part of the JDK. By using the JNI, it ensures that the benchmarking technique is completely portable across all platforms. [18]

The JNI allows Java code that runs within a Java Virtual Machine (VM) to operate with applications and libraries written in other languages, such as C, C++, and assembly. The use of this programming interface meant that some library functions from C such as the library `time.h`, which outputs system-clock time stamps has proved to be significantly helpful.

Chapter 3

Design and Implementation (1st edition)

This chapters illustrates the approach and techniques used to embark on the specifications and objectives set in previous chapter.

The preliminary specification has hence set down the objective, which was to investigate this parallel notion of micro-benchmark and to implement a set of efficient micro-benchmarking applications that will carry out performance prediction of Java programs in a form of bytecode analysis. Through early background research, it has been decided initially to investigate two possible ways of implementing these micro-benchmarks:

- 1. Timing analysis of Java bytecodes**
- 2. Method prediction on Java Programs**

While investigating methods to develop the first type of benchmark, a number of difficulties have been encountered:

1. Finding methods to calculate running time of individual bytecodes / multiple occurrences of a single bytecode to gain a fair timing of the bytecode being tested.
2. Interpretation of the measured timings of bytecodes.

3.1 Finding methods to calculate running time of bytecodes:

When carrying out predictive measurement of testing bytecodes, it is very important that bytecode execution could utilize all the CPU or memory resources available, this means such as invoking a virtual timer, loading in background to measure execution times was not a justifiable option as it could consume CPU and memory resources and compromise the accuracy of predictive measurement, therefore it is more favourable to measure the execution time by finding the time difference of the starting (**st**) and stopping time (**sp**). There are a number of ways this could have been implemented with different degree of accuracy. The following approaches have been investigated:

1. Implement a non-Java program (C, Perl) and invoke the bytecodes sequences by calling the corresponding shell commands e.g. `System("java test")`. The method can be achieved by calculate the running time of the bytecode sequence without the repetition of the bytecode being tested and then apply the same technique to a sequence included with the bytecode being tested.
2. Utilize **System.currentTimeMillis()** method [17], which returns the current time in milliseconds. By invoking the method before and after the repetitive bytecode sequence, the difference in these timings will be the time that takes to execute the testing bytecode repetition. The advantage of this method over the implementation of a non-Java program is that the time method is itself can be expressed in Java bytecode which means all it needs is to be assembled by Jasmin to construct class files.

The disadvantages with these methods are:

Method 1 would have certain overhead that induces inaccuracies. The accuracy of the timing should match the time that took to execute a single bytecode and hence implementing a non-Java program is not the best option.

Method 2 could only produce any bytecode timings to the nearest millisecond, which means it would be also inaccurate for the order of timing that is needed. After several implementation of this method, it was noted that the timing should be in the order of **nanoseconds**.

There was however another method, which has been utilized before, and it had been documented in the technical report of the timing analysis of Java bytecode [4] produced by J.J.Dujmovic at the San Francisco State University. Here is the motivation to consider this new approach.

" A method that executes the bytecode being measured in a controlled context was timed using a Java program and the Unix `clock_gettime()` system call, invoked through the Java Native Interface"

3. Using the idea of **Java Native Interface** (JNI), a C programmed system command which is written to return the time accumulation from 1970 until now as a double value may then be invoked as a Java method before and after the repetitive bytecode sequence, the difference in these timings would be the time that takes to execute the testing bytecode repetition. This method is very similar to method 2 as it can be constructed in Java bytecodes and assembled by Jasmin into class files. The advantage it has over method 2 is that it invokes an external program (written in C), which returns timing in nanoseconds. This will increase accuracy of the testing. [18]

With careful inspection of all three methods, it was reasonable as being supported by another related technical resource that method 3 (using JNI) should be implemented. Nevertheless, the related resources address the predictive measurement by using another assembly language **Oolong**; this is a Java assembler, which uses Jasmin syntax. It has been decided to execute Java bytecodes using the Java Assembler Interface **Jasmin** during specification [1]. Moreover since both Oolong and Jasmin are based on the JVM assembler (JASM), and tests showed they have only very slight syntax differences [2]. It had been decided to utilize both interfaces, As mentioned earlier, Oolong is accompanied by a disassembler interface called **Gnoloo**, which helped the process of implementing a bytecode file at a source-code level. (Gnoloo provide a better disassemble function than javap – class dumper for JDK).

The motivation of this choice meant it was possible to create a Java bytecode-level program file to measure run time of individual bytecodes. The following is an breakdown to show the transformation of an earlier implementation of the benchmark file structure from the source code level (Java) to the bytecode level (Jasmin), note this only demonstrates how the Jasmin description of the benchmark file structure came about by

using assembler and disassembler of the Java Virtual Machine. The actual implementation for benchmarking is explained further in the report.

At Java source-code level shows the structure of how the native methods are invoked in relations to the benchmarking bytecodes:

(Inserting invocation of native method calling the system command clock_gettime())

double start = new ExeTime().displayTime();

.

.

.

Coding being benchmarked...

.

.

(Inserting invocation of native method calling the system command clock_gettime())

double stop = new ExeTime().displayTime();

Listing 3 pseudo-code to illustrate the layout of the benchmarking file at the java source-code level.

Since the structure implemented in Java, by compiling its constituent **.java** source files into **.class** binary, then executing the Java disassembler Gnoloo on these binary files, the above extract could then be executed at the bytecode level using Jasmin: (note: “;” is the syntax to insert comment in Jasmin)

The comments listed indicates the functionalities of each section of the benchmarking structure. These sections are notably the key elements of the benchmark model which is to be formally defined.

```
; <Begin Instantiating ExeTime object>
new ExeTime
dup
aload_2
invokespecial ExeTime/<init>([I)V
astore_3

; <End Instantiating ExeTime object>
; < Inserting invocation of native method calling the system command clock_gettime()>
aload_3
invokevirtual ExeTime/displayTime()D
dstore 4
.
.
.
bytecode being benchmarked...
.
.
.

; < Inserting invocation of native method calling the system command clock_gettime()>
aload_3
invokevirtual ExeTime/displayTime()D
dstore 5
```

Listing 4 pseudo-code to illustrate the layout of the benchmarking file at the java bytecode level.

Since JVM instruction set is a stacked based intermediary language that uses a local stack for its Java method [1], its instructions would involve the manipulation of one or more stack operation (push or pop). Hence this has allowed a **Bytecode Prediction Template** to be developed for benchmarking the instruction set. Diagram 1 defines units or components of the Bytecode Prediction Template.

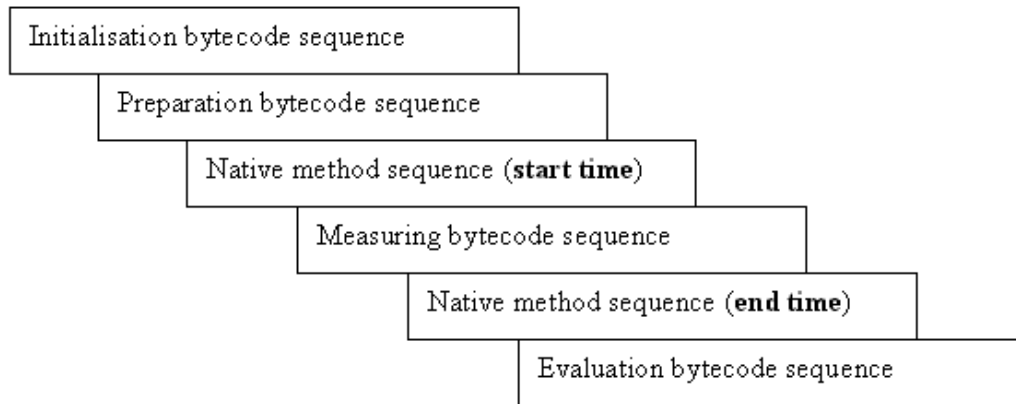


Diagram 1 The Bytecode Prediction Template

Since the instructions that were to be benchmarked would require either one or more elements to be resided on the top of the stack, which is local to the Java method that has been invoked in (e.g. `astore_3`) or values to be assigned to a local variable (e.g. `aload_3`) before they could be invoked, therefore the bytecode prediction template has been formally defined with the components shown above.

3.2 From Template model to implementation

As described earlier, the implementation of the template model could be written in Jasmin (Java Assembler). With such decision in mind, it is important to detail the meaning and the functionalities of each component prior discussing their physical connection with the toolkit that were implemented.

3.3 Components' detail

- **Initialisation bytecode sequence:** -

This component is defined to allow the implementation to initialise. These initialisations happen in all Java programs when they are compiled and executed. Whereas the conventional “source code to machine code” level will disguise such operation, when exercising at bytecode level, so-called Java object initialisation has to be invoked.

Below shows the bytecode implementation of this component.

```
;initialising the native method library
.method static <clinit>()V
    .limit stack 11
    .limit locals 0
    ; loading library with constant "time. This in effect calls library libtime.so
    ldc "time"
    invokestatic java/lang/System/loadLibrary(Ljava/lang/String;)V
    return
.end method

;initialising the Object ExeTime, this is where the template
;implementation is encapsulated
.method public <init>()V
    .limit stack 11
    .limit locals 1
    aload_0
    invokespecial java/lang/Object/<init>()V
    return
.end method

;Declare the native method displayTime()
.method public native displayTime()D
.end method
```

Before executing ExeTime.class, the library path is needed to be defined:

```
export LD_LIBRARY_PATH=~/.jni
for the pathname of the library file libtime.so ~/.jni/libtime.so
```

Listing 5 Bytecode implementation of the initialisation bytecode sequence.

This component is static within the prediction template since all benchmarking processes are executed within the object ExeTime and requires the JNI library **libtime.so** hence almost certain that these bytecodes will be invoked. The only part of this component might be dynamically implemented is object variable definition and method definition for benchmarking bytecodes such as defining static variable for bytecode *getstatic* and defining a static method for bytecode *invokestatic*.

- **Preparation Bytecode Sequence :-**

This is one of the dynamic components of the prediction template. It is defined to act as the initialisation for the **measuring bytecode sequence**. It should provide bytecode sequence so that it recreates the states of the both local stack and local variables syntactically correct prior the execution of the **measuring bytecode sequence**.

- **Measuring Bytecode Sequence :-**

This is also one of the dynamic components of the prediction template, this is where measuring bytecodes will be situated. There are a number of procedures that will be needed to be taken into account when building this component.

1. Allocation of local stack elements.
2. Defining local both local and global variables.
3. Modify bytecodes into executable sequences **
4. Handling redundancy on local stacks.

** Such modification is important for the success of any benchmarking sections. This is because even the majority of bytecodes do not take any argument, many of them not only requires preparation bytecode sequence, they also require a specification of an argument for themselves. Here are some examples:

`iload` – to push an integer value onto the local stack, there is the need to specify local variables that has the integer value, i.e. `iload <varnum>`.

`getstatic` – to get a value of static field, such bytecode requires the specification of a static field and it has a syntax of `getstatic <field-spec> <descriptor>`.

For the duration of this project, this procedure of modification is carried out manually as the implementation for an automated modification is beyond the remit of this project.

- **Evaluation Bytecode Sequence :-**

This is the last component of the prediction template sequentially. Its primary aim is to calculate the timing the measuring bytecode sequence takes to be executed and project the result according to the number of iteration to standard output to which it could then be collected. For parts of this component is dynamically implemented. Its structure in bytecode format is as follow:

It first calculates the timing for single bytecode timing. Note the <number of times> is allocated dynamically as the measuring bytecode is being decided.

```
ldc2_w < number of times>
ddiv
```

```
;keep a copy of original
dstore 3
```

The following bytecodes project the results onto standard output according to the number of iterations. So the outputting data would include the instruction being benchmarked, followed by the number of iterations it has been executed and the timing for a single instance of those iterations.

```
getstatic java/lang/System/out Ljava/io/PrintStream;
new java/lang/StringBuffer
dup
```

```
ldc "<measuring bytecode> , "
invokespecial java/lang/StringBuffer/<init>(Ljava/lang/String;)V
```

```
; <include number of repetition ldc " , ">
```

```
ldc " , <number of times> , "
invokevirtual java/lang/StringBuffer/append(Ljava/lang/String;)Ljava/lang/StringBuffer;
dload 3
invokevirtual java/lang/StringBuffer/append(D)Ljava/lang/StringBuffer;
```

```
invokevirtual java/lang/StringBuffer/toString()Ljava/lang/String;
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
return
```

Listing 6 The bytecode implementation of the evaluation bytecode sequence.

Screen 1 shows a demonstration of an experiment for bytecode multianewarray which requires two arguments and the result is being projected onto standard output.

```
$ java ExeTime  
multianewarray [[I 2, 1, 117120  
$
```

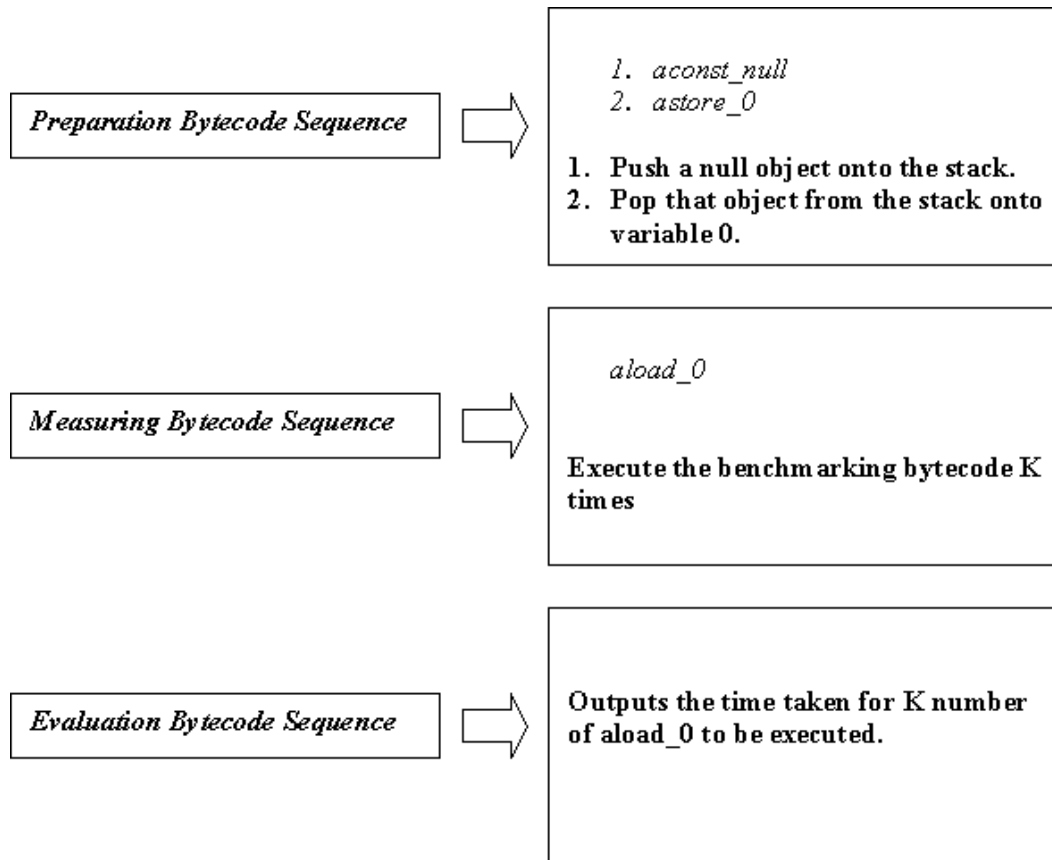
Screen 1 benchmarking bytecode multianewarray

The timings of the execution are given in nanoseconds

3.4 Mechanics of the Bytecode Prediction Template

Below is a diagram representation of how a bytecode is benchmarked mechanically, in this example bytecode `aload_0` is used.

This bytecode pushes an object from variable 0 onto the stack so there is a need to prepare the template before executing the benchmark.



Listing 7 shows how a bytecode is benchmarked mechanically.

From the model to actual implementation it is vital to have the knowledge of the coding and syntax of the template implementation to enable to have a better understanding of how the Perl-written toolkits interact and hence perform their functionalities on the template implementation. The following are descriptions of how the Jasmin implementation of the template interacts with the toolkit written in Perl for processes such as insertion etc.

1. **Preparation Bytecode Sequence** – the comment “*Preparing testing*” has been written in the Jasmin file as a marker to allow the toolkit to be able to identify the exact location to insert preparation bytecode sequences. The use of \diamond is to ensure the toolkit is able to parse the preparation code

```
;Preparing testing  
;<preparation code 1...>  
;<preparation code 2...>  
;...
```

2. **Native Method Sequence** – the comment “*Begin Timing*” and “*End Timing*” signifies the beginning of the section that is static to the toolkit. Certain components of the Bytecode Prediction Template such as this one are consistent throughout benchmarking bytecodes due to their functionalities. Another one of these static component is **Evaluation Bytecode Sequence**.

```
;Begin Timing  
...Native method call  
.....  
  
;End Timing  
... Native Method call  
... ..
```

3. **Measuring Bytecode Sequence** – the comment “*Testing area*” signifies the beginning of measuring bytecode sequences. Similar to the **preparation bytecode sequence**, the use of <> is to ensure the toolkit is able to parse the measuring bytecode.

```
;Testing area  
;<testing bytecode 1...>  
;< testing bytecode 2... >  
;...
```

4. **Evaluation Bytecode Sequence** – There are two areas of this component that interact with the toolkit. Throughout the benchmarking process, either increasing or decreasing the number of iteration is needed and hence the same number as the iteration must be provided to:

- a. Calculate the duration of a single measuring bytecode.

```
;find single bytecode timing  
;<ldc2_w>  
;<ddiv>
```

- b. Be projected with the timings onto the standard output.

```
;<include number of repetition ldc ", ">  
ldc ", 0 times, "
```

In these cases the bytecodes above offers the areas in the template for toolkit interaction.

3.5 Toolkit Development

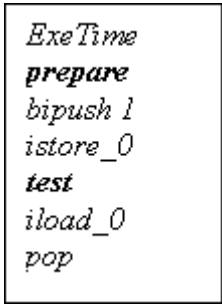
To help in increasing the efficiency of the running of the testing process, the following toolkit written in Perl [3] have been implemented:

- **create_j.pl** - to create the Jasmin file (.j) for a particular bytecode timing sequence.

Usage: - `./create_j.pl <file|filename> <bytecode_name>`

This script requires the following files:

bytecode_name.log - the **template data file** containing preparation bytecode sequence and measuring bytecode sequence.



```
ExeTime
prepare
bipush 1
istore_0
test
iload_0
pop
```

Screen 2 template data file for bytecode iload

Highlighted area from Screen 2 depicts the content of the template data file for bytecode iload, notice the use of keyword such as **prepare** and **test** to identify where these bytecode will be inserted into the **prediction template file** to be benchmarked.

ExeTime.j - the **prediction template file** containing the remaining components of the **bytecode prediction template**. This file is named *ExeTime.j* by default as throughout the benchmarking procedure, the bytecode prediction template is contained within the ExeTime class. Apart from being amalgamated with data from the template data file, this file is fixed as it contains coding that is required for all bytecode prediction.

Below is an extract of the **create_j.pl**. This shows the mechanism of how the script reads the data from the template data file.

```
## Extract preparation code from the template data file ###
if ( $data[$i] =~ m/prepare/ ) {
    $i++;
    while ( ! ( $data[$i] =~ m/test/ ) ) {
        push(@prepare,$data[$i]);
        $i++;
    }
    $flag = 1;
}

## Extract benchmarking code from the template data file ###
if ( $data[$i] =~ m/test/ ) {
    $i++;
    while ( $i < scalar(@data) ) {
        push(@test,$data[$i]);
        $i++;
    }
    $flag = 1;
}
```

Listing 8 shows the mechanism of how the script reads the data from the template data file.

Later on, the default prediction template file have incorporated onto the actual script to save time for file access. Screen 3 depicts the process of inserting these bytecodes.

```
$ ./create_j.pl file byte_sample/pop.log
Insert preparation code
Insert measuring code
$
```

Screen 3 Inserting preparation bytecode sequence and measuring bytecode sequence.

Also as it was decided to test each instruction for 1,10,100,1000 and 9000 occurrences, this was because the measurements of a single bytecode timing were in the range of nanoseconds, and because of this a small fluctuation of CPU resource allocation due to overheads (there would inevitably be background process running within the operating system). These uncertainties would magnify relative to a single bytecode execution time, therefore multiples of bytecode sequences were used and hence the set of timings obtained from each individual bytecode could then be used to make comparison of

accuracy and checks for optimisation and overhead induced by the virtual machine and the architecture. To allow these testing to be carried effectively (as each set of bytecode tests for a single bytecode would require the same template data file), the following two scripts are written to once again increase the speed of testing.

- **increment_j.pl** – takes integer N at shell prompt and inserts the testing bytecode N times in the measuring bytecode sequence and N times the corresponding preparation bytecode sequence. This is also the file of which bytecode measurement would be carried out.

Usage:- `./increment_j.pl <number_of_iteration>`

```
$ ./increment_j.pl 100
Insert prep args 100 times
Insert test args 100 times
$
```

Screen 4 shows the standard output of the Execution of script `increment_j.pl`

Screen 4 is an interface display of the script invocation for measuring a bytecode at 100 occurrences:

As described earlier on the use of `<` to ensure the toolkit is able to parse both the preparation and measuring code. Listing 9 and 10 provides a more detail description and code extracts of how this script insert test bytecodes:

There are three stages in the execution of inserting test bytecodes, the Perl code represents the logic of the code rather than the actual implementation of the toolkit:

N.B. this script has adopted a strategy of which the prediction template file is parsed onto an array and a new array is created by modifying the old array and push its elements onto the new array. This new array is then write onto the file ExeTime.j.

1. Calculate the size of the local stack and the number of the local variable.

```
# this is to ensure that if there is N iterations, the size of stack is at least N#
if( $arg02 < 10 ) {
    $multiple = 10;
} else {
    $multiple = $arg02;
    $multiple = $multiple + 1 - 1;
}

## As the size of stack space does not affect the benchmarking result ##
## the stack size is to be 110% of N ##
$extra = $multiple / 10;
$multiple = $multiple + $extra;
```

2. Capture and increment preparation bytecodes, note the whole prediction template file has been assigned onto the array @file_string.

```
## This is where the script captures the comment "Prepare testing".
if( $file_string[$j] =~ m/Preparing testing/) {

    ## This initiates the gathering of preparation code.
    ## This condition catches the beginning of <>
    while ( $file_string[$j] =~ m/<\/ ) {

        ## store everything after "<"
        ## delete ">" to obtain the prediction code.
        $prep_string = $';
        $prep_string =~ s/>\s+//g;

    }
}
```

Listing 9 description and code extracts of how the script increment_j.pl insert test bytecodes

3. Capture and increment testing bytecodes, the similar methodology to that of incrementing prediction bytecode is used to carry out this process.

```
## This is where the script captures the comment "Testing area".
```

```
if ( $file_string[$j] =~ m/Testing area/ ) {
```

```
    ## This initiates the gathering of testing code.
```

```
    ## The remaining code for collecting bytecode sequences is
```

```
    ## identical to that of incrementing prediction bytecode
```

```
    while ( $file_string[$j] =~ m/;</ ) {
```

4. Modify Evaluation Bytecode Sequence to calculate the duration of one measuring bytecode execution.

```
## This is where the script captures the comment "find".
```

```
if ( $file_string[$j] =~ m/find/ ) {
```

```
    ## This initiates the gathering of evaluation code.
```

```
    ## Insert new data for evaluations
```

```
$num = $file_string[$j];
```

```
$num =~ s/<|>|;/g;
```

```
$num .= " ".$arg02.".\n";
```

```
}
```

```
## This is where the script captures the comment "repetition" and "include".
```

```
if ( ( $file_string[$j] =~ m/repetition/ ) && ( $file_string[$j] =~ m/include/ ) ) {
```

```
    ## This is where the code is modified for the result to be projected
```

```
    ## onto standard output correctly
```

```
push(@new, " ldc \", ".$arg02." times, \n");
```

```
$j++;
```

```
}
```

Listing 10 more description and code extracts of how the script increment_j.pl insert test bytecodes

- **aver.pl** - to further refine the testing procedure, any iterations of bytecode sequences would be tested 10 times and this was decided in making sure it was a fair test and that timings do not have overheads incurred by other programs running elsewhere on the operating system. This script helps to find an average of a particular bytecode sequences from the result.log (ASCII file).¹

Usage:- `./aver.pl <measuring_bytecode>`

This script interacts with all the scripts that were written previously and Screen 5 depicts the result this scripts generates onto standard output.

```
Inserting prep args 100 times
Inserting test args 100 times
Generated: ExeTime.class
Average timing is 175.033333333333
istore looped 100 times is 175.033333333333

Inserting prep args 1000 times
Inserting test args 1000 times
Generated: ExeTime.class
Average timing is 57.4666666666667
istore looped 1000 times is 57.4666666666667

Inserting prep args 9000 times
Inserting test args 9000 times
Generated: ExeTime.class
Average timing is 48.8333333333333
istore looped 9000 times is 48.8333333333333

istore looped 1 times is 13043.2
istore looped 10 times is 1342.61538461538
istore looped 100 times is 178
istore looped 1000 times is 57.7
istore looped 9000 times is 48.5
```

Screen 5 The standard output of the benchmarking of bytecode istore.

The screenshot clear illustrates that each bytecode is executed with interval iterations to sample how efficient the Java Virtual Machine processes the bytecodes and the duration of these individual bytecode against its repetition. Below is a technical description of how such a script is implemented to accommodate the other scripts and hence defines the toolkit.

Note codes in this section are incomplete as it is the logic that is being demonstrated.

```
## This section builds the template file with the specified bytecode sequence
$template="/create_j.pl file byte_sample/"$argument.".log";
system("$template");

## This while loop allows bytecodes to be benchmarked at 1,10,100,1000 iterations
while ($k < 5) {
    if ($k == 0) {
        $test_mult=1;
    } elsif ($k == 1) {
        $test_mult=10;
    } elsif ($k == 2) {
        $test_mult=100;
    } elsif ($k == 3) {
        $test_mult=1000;
    } else {
        $test_mult=9000;
    }
    $k++;
}

## This section compiles the template for a particular iterations.
## Note this section is part of the while loop described earlier
$command = "/increment_j.pl "$test_mult." 3";
$times = $test_mult;
system("$command");
system("/dcs/00/csveel/private/research/jasmin/bin/jasmin ExeTime.j");

## This section executes the template file 10 times.
## Timing is piped into result.log of which will be read to get the average.
$i = 0;
while ($i<10) {
    $i++;
    system("java ExeTime >> result.log");
}
```

Listing 11 a technical description of how such aver.pl is implemented to accommodate the other scripts and hence defines the toolkit.

3.6 Formal Evaluation

As each bytecode would be tested for 1,10,100,1000 and 9000 iteration(s) sequence. The timing of a single bytecode (**ot**) would be calculated as: (measured in nanoseconds)

$$\begin{array}{l} \text{one bytecode time} = (\text{stop-time} - \text{start-time}) / \text{number of iteration} \\ \text{ot} = (\text{sp} - \text{st}) / \text{N} \end{array}$$

As mentioned on earlier documentation, some bytecodes such as an integer load operation (`iload`), which manipulate the local stack by pushing an element or returning a value onto it. However, these would create inconsistency with the stack state for the rest of the Java object and so to make sure no **redundant value** was left on the stack, the stack would be popped by invoking the bytecode `pop` on every occurrence of such bytecode in the measuring bytecode sequence and so the timing in general for this situation would be calculated by taking away the timings of the extra bytecode used in removing redundant value (**et**) : (also in nanoseconds) ¹

$$\text{ot} = ((\text{sp} - \text{st}) / \text{N}) - \text{et}$$

3.7 Interpretation of the measured timings of bytecodes:

As well as part of the specification, it is important to confirm that the technique to benchmark the performance of bytecodes is within an acceptable accuracy. It is therefore important to carry performance prediction on a small Java sequential program which include performance critical section to be benchmarked on.

Such prediction sessions of Java programs and hence matches the results against real-

¹ The timing of these extra bytecodes would be obtained by the same method as the others

time analysis or previous results have been implemented during the course of the project. A Bubblesort algorithm program is used for carrying out these prediction sections.

The main critical section of this algorithm is shown below.

In this extract of Java code, "a" is the unsorted array of integer, since the following parts are visited more than once during the course of sorting the array, these two methods are the critical sections of the algorithm.

```
public void sort() {  
    int temp;  
    // entering outer loop  
    for (int i = 0; i < a.length - 1; i++) {  
        // entering inner loop  
        for (int j = a.length - 2; j >= i; j--) {  
            if (a[j] > a[j+1]) {  
                // entering swap method  
                swap(j, (j + 1));  
            }  
        }  
        // exiting inner loop  
    }  
    // exiting outer loop  
}  
  
// This is the kernel of the BubbleSort Algorithm.  
// This is performance critical.  
private void swap(int x, int y) {  
    int temp;  
    temp = a[x];  
    a[x] = a[y];  
    a[y] = temp;  
}
```

Listing 12 The Java Bubblesort algorithm kernel

The bytecode constituents of these methods with their predictive timing are listed on the appendix repository.

This is the sorting algorithm in the Bubblesort.java that contained two for loops and number of iterations for these loops depended on **a.length**, which is the length of the array that will be sorted in these methods.

The program initially set the size of the unsorted list to be `a.length`. The 1st loop (outer) takes (`i = a.length-1`) where `i` is the pointer for the outer loop, The inner loop of the iteration takes (`j = a.length-i-2`) for every `i`'th iteration where "`i`" is the variable allocated to the first for loop.

To cater the randomness of the array, since not every loop would invoke the swap method, the probability of invoking swap method hence was decided to be 0.5.

e.g. For a unsorted list size of 100

- number of outer loop iteration = 99
- number of inner loop iteration = 4950
- number of times invoking swap method = 2475

Without the use of the evaluation engine in PACE, a more conventional method of summing up the number bytecode values with the number of times they are being invoked was implemented.

- **evaluation.pl** – takes an integer argument that will calculate the required measurement. This script was implemented to accumulate all the bytecode results in relation to each individual method of the sorting algorithm, which are performance critical. This was carried out by scanning through **bs.jPtran.xml**, which is an XML file used to characterise the BubbleSort.java under the Evaluation Engine.

Usage: `./evaluation.pl <number_of_unsorted_item>`

This script requires the following sequential programs and ASCII files:

Run.java – This is a sequential Java application that is designed to return the number of outer loop iteration, inner loop iteration and number of times invoking swap method based on the size of the unsorted array. Screen 6 shows the standard output values from this application running with an array of 150 unsorted element

```
$ ./java Run 150
149, 11175, 5587
$
```

Screen 6 Execution of Run.java with an array of 150 unsorted element.

BubbleSort.java – This is the testing application and it is used in conjunction with `clock_gettime()` to benchmarking the performance critical section of the sorting algorithms. This is appropriate as the benchmarking is carried at source-code level and the same C library function is used as to when benchmarking at bytecode level. Below is a benchmarking section of *BubbleSort.java*.

```
double start = new ExeTime().displayTime();  
sort.sort(); (critical section / sort & swap method are invoked)  
double stop = new ExeTime().displayTime();
```

This application takes the unsorted array size as its argument and returns the time it takes to sort the unsorted array using the Bubble Sort algorithm in nanoseconds.

data.log – this ASCII file contains the benchmark timing of the constituent bytecodes of the sort and swap algorithms. Screen 7 shows the content of *data.log*.

```

lconst_0, 33175.6, -, 3425.5, 384.7, 86.4, 53.1
Aload_0, 32317, -, 3267.1, 365.8, 94.4, 58.9
Arraylength, 9095.7, -, 149.2, 113.32, 112.7, 139.2
lsub, -2694.1, 32710.4, 567.6, 88.62, 74.1, 73.4
lstore_0, 34665.5, -, 3356.3, 382.7, 211, 67
lload_0, -1027.46, 34377.04, 1270.43, 64.15, 60.8, 37.05
ladd, 2551.2, 35678.9, 333.8, 96.7, 67.4, 70.3
invokespecial, -143, 32984.7, 3101.8, 4944.7, 3704.7, 4914.4
lsub, -47.7, 33080, 376.5, 92.7, 66.7, 116.9
linc, 34331.9, -, 3929.8, 398.2, 97.9, 68.2
if_icmpge, 8022.3, -, 544.2, -17, 156.9, 215.5
if_icmple, 7807.7, -, 535.9, -26.3, 216.7, 324.4
Goto, 34117.1, -, 3561.6, 437.7, 139.9, 99.9
if_icmplt, 7235.4, -, 568.1, -16.1, 213.3, 232.2
laload, 1227.9, 34355.6, 242, 264.65, 93.35, 96.47
lastore, 35738.4, -, 3467.5, 459.2, 146.6, 132.1
Getfield, 12588.6, -, 1745.3, 847.3, 711.3, 796.5
pop, 33127.7, -, 3270.6, 358.7, 92.7, 56.6

```

Screen 7 Content of data.log

Listing 11 and 12 are the technical description of the logic of evaluation.pl:

There are four main steps to this process:

1. Execute BubbleSort.java and Run.java to the relevant values.

```

$iteration = "java Run ".$ARGV[0];
$against = "java BubbleSort ".$ARGV[0];

```

Listing 13 first step of the logic of evaluation.pl

2. Extract relevant bytecodes from the XML characterisation file (bsjPtran.xml) and remove the <varnum> from bytecode accordingly. Group the bytecodes into their corresponding loops and methods. Below are the regular expressions or patterns that extract and gather the relevant bytecodes.

```
## method sort()
$xml[$count] =~ m/method=\\"sort\\"/
## Outer Loop
($xml[$count] =~ m/variable=\\"i\\"/) || ($xml[$count] =~ m/jPACE\carryon/)
## Inner Loop
($xml[$count] =~ m/variable=\\"j\\"/) || ($xml[$count] =~ m/jPACE\prob Value/)
## method swap()
$xml[$count] =~ m/method=\\"swap\\"/ && ($xml[$count] =~ m/jPACE\method/)

findopcode($count,@xml)  a subroutine that is responsible to extract individual
bytecodes from the characterisation file (array @xml).
```

1. Extract benchmark timings from data.log and parse data onto associative arrays (hash) with the required bytecodes as keys and there execution benchmark times as values. Below is the line that splits every line of data.log into variables and these variables are assigned to hashes.

```
($name,$onea,$oneb,$ten,$hundred,$thousand,$more) = split(/,/);
```

2. Use the values from step 1, the bytecode groups from step 2 and the predicted timing from step 3 to carry out prediction and comparison on the sorting algorithm using its bytecode constituents.

Below is the formal definition of the equation that calculates the prediction of this sorting algorithm is:

T_x predicted time T of the algorithm with an unsorted array size x
O_x predicted time of the outer loop with an unsorted array size x
I_x predicted time of the inner loop with an unsorted array size x
W_x predicted time of the swap method with an unsorted array size x
S_x predicted time of the sort method with an unsorted array size x
A_i number of times ith bytecode exists within section A
T_i predicted time of ith bytecode
A_{xn} number of times section A executes with an unsorted array size x

$$T_x = O_x + I_x + W_x + S_x$$

Where

$$O_x = \sum O_i \times T_i \times O_{xn}$$

$$I_x = \sum I_i \times T_i \times I_{xn}$$

$$W_x = \sum W_i \times T_i \times W_{xn}$$

$$S_x = \sum S_i \times T_i \times S_{xn}$$

Listing 14 the remaining technical description of the logic of evaluation.pl

The results of these calculations could be found in the appendix. During the course of this investigation, an assumption has been made:

That the time difference of invoking same bytecodes that retrieve and assign values onto different variables local to the method could be neglected.

e.g. iload_2 and iload_3 was assumed to have the same execution time.

Furthering from these bytecodes running, an interesting observation was made on the timing measured with one bytecode iteration, some timings were negative and this was because when certain bytecodes have been executed, a redundant value or a redundant object reference might be left on top of the local stack and therefore pop was invoked after each occurrence of these kinds of bytecodes to ensure the program operates without the interference of the data manipulation from the testing bytecodes.

Since the timing of pop has been recorded and so there might be some discrepancies in the measurement of bytecodes at different instance. It was thought that by just measuring one bytecode iteration might lead to a bigger inaccuracy.

3.8 Observation:

The timing came from the evaluation script did not exactly match the time measured from the BubbleSort.java 's performance critical section. The reason although not obvious, it was understood that JVM would carry out optimisation and since the structure of the sorting algorithm meant that same bytecodes would have been invoked as many as the number of iterations (in fact for 100 elements in an unsorted list, 4950 iterations of the inner loop would be invoked). This meant that there is a need to investigate the discrepancies due from optimisation or otherwise.

This could be one of the reason (and the same reason) as to why when individual bytecode was timed, one iteration took more time than an average of multiple iterations (e.g. 1000).

Below is a list of points that is needed to be considered:

- Bytecode latency from invoking native method (calling C library).
- Effects of Hot Spot compiler or Java Optimisation.
- Speed difference between invoking same bytecodes that retrieve and assign values onto different variable.

Chapter 4

Development

This chapter formally discusses the observation after the initial analysis of the predicted execution time and measured execution time of the Java Bubblesort Algorithm kernel. It details the modern Java Hot Spot™ optimisation and techniques used to overcome the inaccuracy caused by this technology.

4.1 Initial thoughts and experiments:

These areas, which might have caused the inaccurate benchmark timings, were examined and consequently their significances in the accuracy of the timings were decided:

4.2 Bytecode latency from invoking native method

Below is a diagrammatic representation of the Java Native Interface [18]

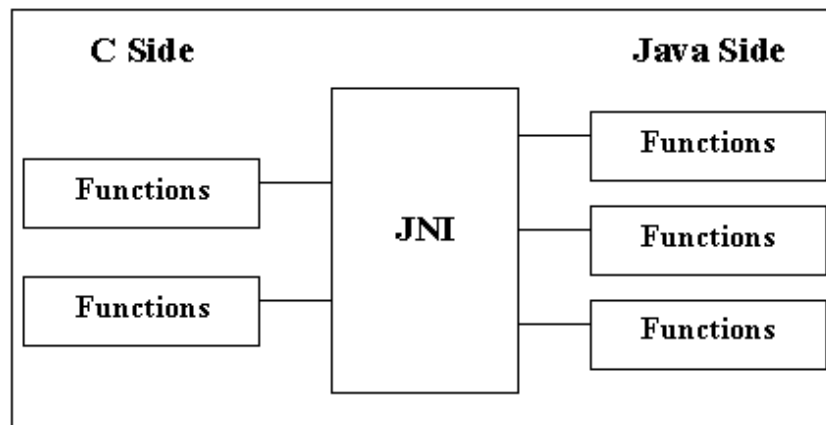


Figure 7 Diagram depicts the mechanics of JNI

From the diagram that latency could be caused by the “C side” where the *clock_gettime()* is executed. Below is the implementation of the Native Method, note the function of *clock_gettime()* function:

```

#include <jni.h>
#include "ExeTime.h"
#include <stdio.h>
#include <time.h>

JNIEXPORT jdouble JNICALL Java_ExeTime_displayTime (JNIEnv *env, jobject obj)
{
    struct timespec t;
    clock_gettime(CLOCK_REALTIME, &t);
    return ( (t.tv_sec * 1e9 ) + (double)t.tv_nsec );
}

```

Listing 15 the implementation of the Java Native Method

It could be seen the implementation has been written in the simplest format. Through further research it has been shown that the latency from native methods was insignificant and hence it could be neglected.

4.3 The latency in variable assignment and retrieval

When interpreting the results, the following assumption has been made:

That the time difference of invoking same bytecodes that retrieve and assign values onto different variables local to the method could be neglected.

This assumption might have led to the inaccuracy of benchmark timings. As formally described, each method invocation has its own set of local variables. Local variables hold the formal parameters for the method. Technically, it is thought by keeping more frequently used values in lower-numbered local variables may improve performance. To analyse this situation below is a diagram illustrates what might happen when a local method retrieves a value from a local variable.

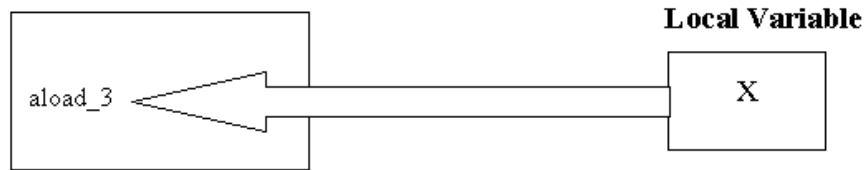


Diagram 2 a diagrammatic illustration of pushing a value from a local variable onto the local stack.

This is a standard operation to retrieve X from local variable 3 and push it onto the local stack. This might look rather trivial but let's illustrate a hypothetical situation:

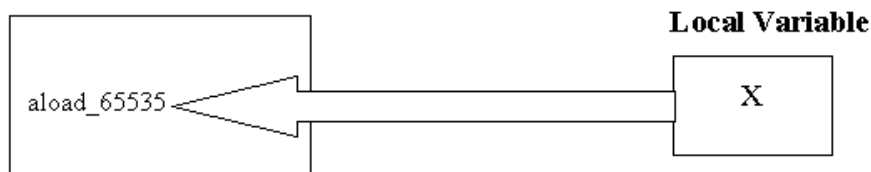


Diagram 3 a diagrammatic illustration of pushing a value from a high-numbered local variable onto the local stack

Now even though the probability that a method is to manipulate the local variable 65535 is quite minimal, it is a good method to analyse the need to take the position of these local variables into account. If these positions are being mapped onto a hardware configuration then the time it takes to manipulate a lower-numbered variable should be less than a high-numbered variable mainly due to the physical position of these variables on the hardware registries. This is also a valid argument when viewing the Java Virtual Machine as a virtual processor.

With the implementation of the characterisation XML file such that it has become inefficient to have to first identify which the exact local variable numbers are to be processed before carrying out bytecode benchmark. A good example is to compare such manipulation with the invocation of a bytecode instruction that manipulates an array structure, e.g. `iaload`. It would not be suitable to pinpoint the index of the array that this bytecode would be used in the application that is to be benchmarked as the index of the array is most likely to be dynamically allocated in the application depending the stages of execution. And yet it is important to know the index it has been assigned with bearing in mind the physical location of each element of the array. If the same analogy is applied back to the manipulation of local variable then such inadequacy it can be immediately

seen. Therefore to ensure future benchmarking is to be fair the following restriction has been imposed:

When allocating variable numbers to variable manipulating bytecodes e.g. `aload`, it should be carried out systematically and consumed the lowest-numbered variable available first.

4.4 The effects of Hot Spot compiler and Java Optimisation.

Assumption:

- If optimisation was regular and predictable then theoretically by running single bytecode n occurrences in a sequence, **$n-1$ occurrences of them will be optimised.**

This meant that if:

Timing of one bytecode x (raw / as one occurrence): t

Timing of n occurrences of bytecode x : s

***Timing of one bytecode x (optimised):* $OT = (s-t) / (n-1)$**

However, it was thought that due to the sophisticated implementation of the Java Virtual Machine, there are progressive optimisation within the compilation and execution of Java application, hence such assumption was not suitable for revised implementation.

4.5 Hot Spot Motivation

In the past, most attempts to accelerate Java programming language performance have focused on applying compilation techniques developed for traditional languages. Just-in-time (JIT) [21] compiler is an example that is essentially a fast traditional compiler that translates the Java technology bytecodes into native machine code on-the-fly. A JIT compiler runs on the end-user's machine and actually executes the bytecodes, compiling each method the first time it is executed. JIT compilations has included a selection of optimisation toolkits and they provide the following functionalities:

1. **Base JVM modifications** - There are major changes introduced to improve the overall performance of the `JIT` compiler: a change in the object layout and the execution of the static initialiser. First the change in the object layout for both ordinary objects and array objects. This change allows direct access to instance fields simply by adding an extra offset to the object pointer, This is a great advantage in terms of code generation efficiency, since the array bound exception checking has to be done every time an array element is accessed. In terms of the execution of the static initialiser, the resolution of a class has been separated from the execution of its static initialiser, By separating the class resolution and the execution of its static initialisation, the `JIT` compiler has more opportunity to generate faster code, using run-time calls if necessary to run the static initialiser.
2. **Selective Compilation** – Since `JIT` compilation occupies a part of the application run time; it is not necessarily beneficial to compile all the methods being invoked. For example, when a method is executed only once and does not contain any loops, the overall performance might be degraded if it is `JIT`-compiled. The cost of the `JIT` compilation needs to be offset by the performance gain achieved by running the native code in terms of both time and space. Therefore by adopting an appropriate way of identifying and choosing “hot” methods that deserve `JIT` compilation, it is expected to achieve high performance in running real applications as well as benchmarking programs.

Unfortunately, these kinds of compilations surface several subtle problems. [19]

1. Since the compiler runs on the execution machine in "user time," this means that its compiling speed is severely constrained: if it is not very fast, then the user will perceive a significant delay in the start-up of a program or part of a program. This imposes a trade-off that makes it far more difficult to perform advanced optimisations, which usually slows down compilation performance significantly.
2. With the problems imposed by Java Virtual Machine's advanced functionalities such as garbage collection that causes more memory allocation overhead than the conventional C++, and the on-the-fly changes through the ability to perform dynamic loading of classes which hinders the performance of many types of global optimisation. These problems suggest that even if a JIT compiler had time to perform full optimisation, such optimisations are less effective for the Java programming language than for traditional languages like C and C++.

This results in the incapability of conforming to any traditional compiler techniques to achieve advances in Java programming language performance. The Java HotSpot VM architecture addresses the Java programming language performance issues by using adaptive optimisation technology.

4.6 Hot Spot Detection

Adaptive optimisation solves the problems of JIT compilation by taking advantage of an interesting property of most programs. Virtually all programs spend the vast majority of their time executing a small minority of their code; this is very much consistent with the idea in the initial design to concentrate in only performance critical section of sequential programs.

Therefore, instead of compiling sequential programs method-by-method, just in time, which is the original intent of JIT compilation, the Java HotSpot VM runs the program immediately using an interpreter and analyses the code as it runs to detect the critical "hot spots" in the program. It then focuses the attention of a global native-code optimiser on the hot spots. By avoiding compilation of infrequently executed code (most of the program), the Java HotSpot compiler can devote much more attention to the performance-critical parts of the program, without necessarily increasing the overall compilation time. This hot-spot monitoring is continued dynamically as the program runs, so that it literally adapts its performance on-the-fly to the needs of the user.

A subtle but important benefit of this approach is that by delaying compilation until after the code has already been executed for a while ("a while" in machine time, not user time), information can be gathered on the way the code is used, and then used to perform more intelligent optimisation. Also, the memory footprint is decreased. In addition to collecting information on hot spots in the program, other types of information are gathered, such as data on caller-callee relationships for "virtual" method invocations.

Moreover, since the frequency of virtual method invocations in the Java programming language is an important optimisation bottleneck. Once the Java HotSpot adaptive optimiser has gathered information during execution about program hot spots, it not only compiles them into native code, but also performs extensive method inlining on that code. Inlining has become more important than before as inlining produces much larger blocks of code for the optimiser to work on, significantly increasing the effectiveness of

traditional compiler optimizations, and thus overcoming a major obstacle to increased Java programming language performance.

There are also other features that may concern the prediction of JVM instructions and one of the main features is Dynamic deoptimisation:

4.7 Dynamic de-optimisation

Although inlining is an important optimisation, it has traditionally been very difficult to perform for dynamic object-oriented languages like the Java programming language. Furthermore, while detecting hot spots and inlining the methods they invoke is difficult enough, it is still not sufficient to provide full Java programming language semantics. This is because programs written in the Java programming language cannot only change the patterns of method invocation on-the-fly, but can also dynamically load new Java code into a running program.

At the bytecode level, the interpreter in Sun's Java Development Kit reference implementation does inline some simple methods, if the bytecode they contain fits into the space for method invocation or converts the calls to empty constructor methods to `invokeignored_quick` instruction. Such inlining is based on a form of global analysis. Dynamic loading significantly complicates inlining because it changes the global relationships in a program. A new Java class may contain new methods that need to be inlined in the appropriate places. So the Java HotSpot VM must be able to dynamically deoptimise (and then reoptimise if necessary) previously optimised hot spots, even during the execution of the code for the hot spot. Without this capability, general inlining cannot be safely performed on Java technology-based programs.

Hot Spot Detection and other known enhanced optimisations meant that the previous notion in benchmarking implementation would not be accurate and it was vital to know how Hot-Spot detection mechanically structured so that the revised implementation of bytecode monitor can address prediction that is comparable to real-time execution.

There are also other well-known optimisations such as Just-In-Time (JIT) Compilation, which has been mentioned, that are necessary to be taken into account of. The current version of JIT includes a repository of set of common bytecode sequences (CBS). This notion has already been discussed and established in the Progress Report at the end of WK 10 Term 1. Unfortunately the detail of this repository is not known and hence rather than implementing CBS, another concept has been adopted instead

4.8 Developed Ideas

By detailing and locating areas that might constitute the inaccuracy of the benchmark timing, the following developed idea has been laid down:

These observations suggested that more emphasis should be laid on JVM optimisation; some manipulations of results from the predictive measurement were carried out and with a better understanding of the optimisations from Java Just-In-Time and Hot-Spot compiler, it has become apparent that a new notion of analysis can be initiated. The motivation, similar to the idea of **Method prediction on Java Programs**, was instead of analysing Java bytecode individually in repetitions, the execution time of **blocks of sequential bytecodes** could be examined. These common sequences of bytecodes are optimised as a unit.

Previously, it is because the optimisation of the JIT compiler is implemented by parsing blocks of common bytecode, therefore by analysing blocks of bytecodes that the logic of JIT optimisation can be extracted and processed, a database of common bytecode sequences can be created. Furthermore, gaining knowledge of this logic can also assist the analysis of the uncertainty within the execution time of bytecode that were obtained from previous experiments. However, the current release of JIT compiler of which its logic includes method inlining, base JVM modification and selective compilation is beyond the time frame of this project. Moreover it is not possible to obtain an accurate logical implementation of the JIT compiler due to business confidentiality and also the

machines, of which prediction experiments are conducted on favour the use of HotSpot Compilation technology over JIT.

Consequently, with the current available resource and time limit, HotSpot Compilation and optimisation is taken into account for a revised design.

4.9 From ideas to Implementation

To implement a new benchmark system with bytecode sequences rather than single bytecodes, first there is a need to understand how HotSpot optimisation works at the source code level. This can be done with illustrations of examples. [15]

The following benchmark typifies a simple benchmark that doesn't benefit from HotSpot technology:

```
public class SimpleBenchmark {
    public static void main(String[] argv) {
        int value=0;

        // Record the start time.
        long start= System.currentTimeMillis();

        // Repeatedly executes feature to measure performance.
        for (int i=0; i<10000000; i++) {
            // Replace line with your favorite computation.
            value +=i;
        }

        // Record the finish time.
        long finish= System.currentTimeMillis();

        // Now report how long test ran.
        System.out.print ("Time spent = " +
            Long.toString(finish - start) + " ms\n");
    }
}
```

Listing 16 A section of Java code being benchmarked (high level)

As HotSpot compiler selectively converts Java bytecode dynamically into highly-optimised machine instructions. The overhead for such a compiler is higher than for a JIT. It performs analysis on each application in order to identify the most frequently used areas. After the program's "hot spots" have been identified, these sections of code are compiled and optimised.

The current implementation of HotSpot is designed for long-running applications. Most applications spend the majority of their time executing a small section of their code. These sections are known to be performance critical in the field of High performance computing. This paradigm is referred to as the 80/20 rule where, as a generalization, programs spend 80% of their time executing 20% of their code.

HotSpot initially runs the Java application in interpretive mode while it analyses the application for "hot spots". This optimisation consists of compiling and in-lining critical methods to achieve optimal performance. After the "hot spots" have been identified and optimised, HotSpot will then switch from executing interpreted bytecodes to executing the corresponding compiled code. This analysis-and-optimisation impacts performance. Longer-running applications will benefit more from HotSpot optimisation because they run longer and will be executing the compiled code longer. These programs can afford the temporary performance impact associated with analysis and compilation.

The crucial part for an accurate benchmark, which is efficient it is essential to understand how HotSpot converts from executing interpreted-bytecodes to compiled code. Whilst HotSpot detects performance critical areas and converts them into compiled code, the compiled version of the code is not invoked until the next time the method is called. Thus, if the method is only called once, such as from `main()`, then optimisation will not take place. This means that the program pays the price for analysis and optimisation that will never be used.

Another issue that is needed to be accounted of is the performance loss that is caused by benchmarking small amounts of code for only very few iterations. This is because the benchmark would be finished before optimisation begins. Thus for short-term applications, the cost of using HotSpot technology is actually more of a performance detriment because it must analyze the application before compiling any code. However, real-world applications tend to not to be small applications, especially High performance application.

These consideration leads to a slightly different implementation of the benchmark that will take the advantage of HotSpot technology.

```
public class HotSpotBenchmark {
    public static void runTest() {
        int value=0;
        // Repeatedly executes feature to measure performance.
        for (int i=0; i<10000000; i++) {
            // Replace line with your favorite computation.
            value +=i;
        }
    }

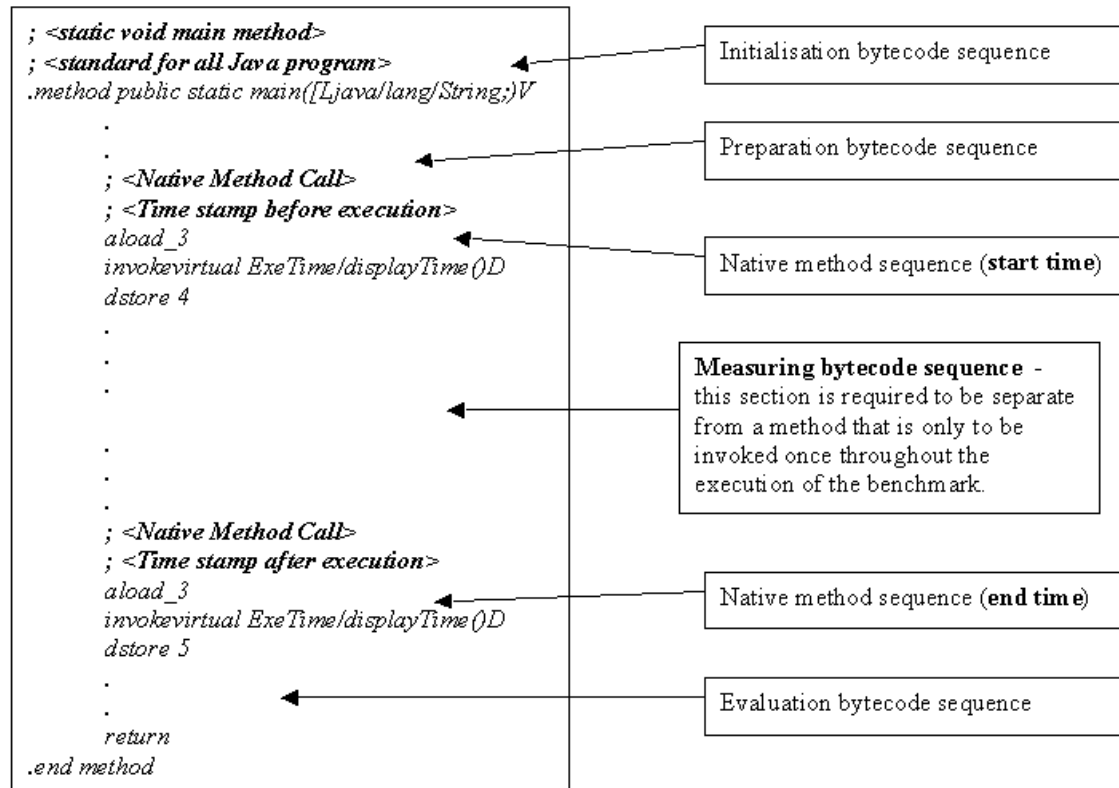
    public static void main(String[] argv) {
        // Run benchmark multiple times. This will allow us to
        // see when HotSpot begins executing compiled code.

        for (int i = 0; i < 8; i++) {
            // Record the start time.
            long start= System.currentTimeMillis();
            // Run benchmark test.
            runTest();
            // Record the finish time.
            long finish= System.currentTimeMillis();
            // Now report how long test ran.
            System.out.print ("Time spent = " +
                               Long.toString(finish - start) + " ms\n");
        }
    }
}
```

Listing 17 Revised implementation of Java code for performance benchmark

Note the HotSpot technology executes the interpreted version of the method several times before running the optimised version.

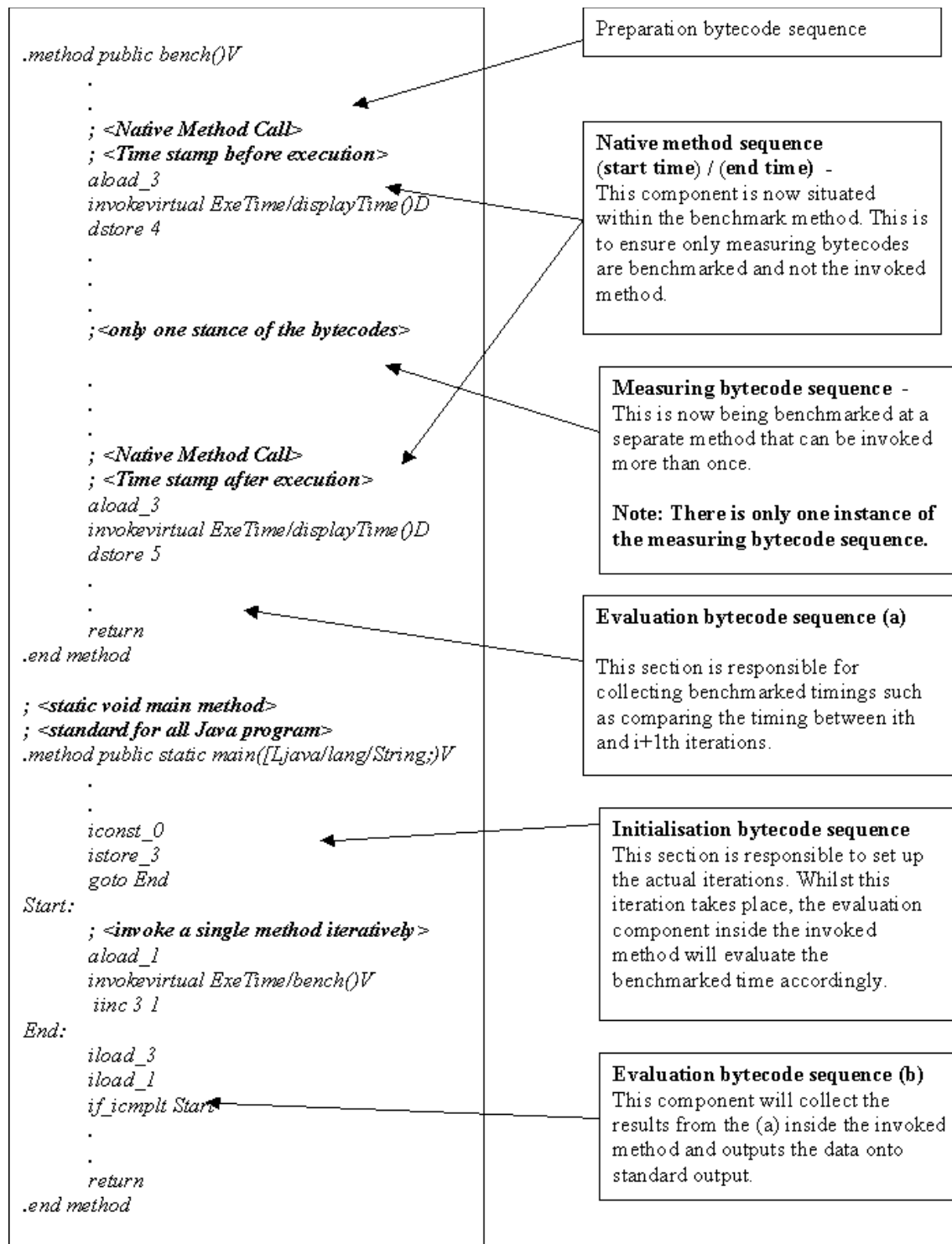
From the run down description of how HotSpot optimisation could alter the performance a small extract of Java sequential application and the methodology that caters this technology, the same concept can also be applied to benchmarks at bytecode level. Below shows describe how the previous design should be changed to adapt HotSpot optimisation.



Listing 18 A model that has been adapted to include Hot Spot™ technology and method level optimisation

From above it became apparent immediately that the original structure, which benchmarks bytecodes inside a method that is to be called only once, namely `main()` did not allow optimisation to take place as the compiler would not be able to detect any HotSpot even if there are multiple instances of the same bytecode being executed consecutively under the same class.

To effectively account for the effect of optimisation it is essential for the **measuring bytecode sequence** from the **bytecode prediction template** to be invoked within a separate method that is to be called from the main method. This can be thought to be similar to `runTest()` from the previous source code illustration. However whereas at source code level the whole method invocation is benchmarked, at bytecode level this is clearly not the case. Below is the revised model that had been decided when trying to compromise an optimal solution with the bytecode prediction template.



Listing 19 section 2 of the model that has been adapted to include Hot Spot™ technology and method level optimisation

Previous page illustrate the new bytecode prediction template, note the benchmark timestamp occurs inside the invoked method `bench()` and the iteration takes place outside the benchmark timestamp. The reason for the revised model to include these features is that to allow a firm compatibility with the original prediction template, which benchmarks multiple instances of the measuring bytecodes and extracts the timing for a single measuring bytecode sequence. This revised model instead, only ever benchmarks one instance of measuring bytecode sequence, and the evaluation sequence described will compare the timings from each iterative instance and outputs the data onto standard output accordingly.

Chapter 5

Design and Implementation (2nd edition)

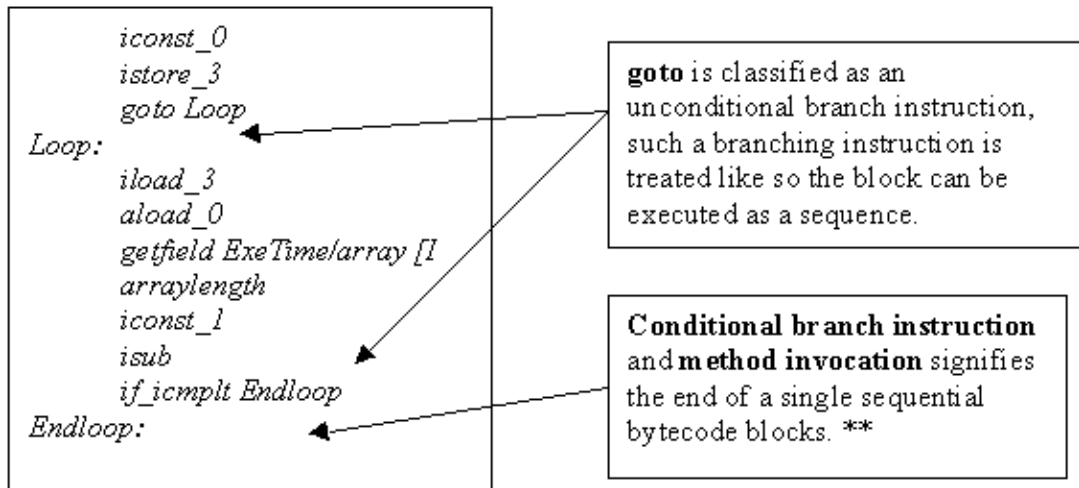
Java Hot Spot™ technology allowed Java applications to be optimised on-the-fly and since Hot Spot detection is carried out at method level, a new design and implementation is realised and this chapter discusses the technique of this bytecode monitor and illustrates its technical details

With the observation and insight gained from the initial implementations and results, it has been decided to revise the atomic unit of performance within the characterisation language, and such needs for Hot Spot compiler optimisations means that Java methods are now characterised as a control flow of bytecode blocks, rather than individual bytecodes. This is because it has been suggested that Hot Spot is likely to be a set of sequential instructions rather than one bytecode operation, since this is the case it is not necessary to benchmark individual bytecodes when optimisation only takes place with bytecode blocks.

These bytecode blocks are then benchmarked, and it is their timings that are used when obtaining predictions. However, these timings vary depending on whether they have been optimised during execution, and so this is also taken into account during model evaluation. This is explained in more detail below. [5]

5.1 Bytecode Block Definition (Sequential Bytecode Block)

Currently, bytecode blocks are defined as sequences of bytecodes that do not contain any conditional branch instruction or method invocation opcodes. Below is an example of a bytecode block in the BubbleSort.java. It should illustrate some distinct features of the bytecode block definition (SBB).



Listing 20 shows some distinct features of the SBB

Conditional branch instruction and method invocation define the end of each **SSB. Although method invocation is not conditional, it cannot be included as a part of another bytecode sequence. This is because in the revised version of the XML characterisation file for PACE, each method and object class are being characterised separately.

This means that bytecode blocks can be of varying sizes, from only one opcode to theoretically the maximum limit of the size of a method permitted by the JVMspecification1. A method that only contains a set computation without any loop or conditional statements and does not invoke any other methods will be characterised as one bytecode block.

A tool similar to ‘capp’ has been developed that parses Java class files and outputs an XML-based performance characterisation of the appropriate bytecode. The tool uses a method very similar in approach to that of a class file decompiler, extrapolating ‘for’, ‘while’, ‘if’, ‘switch’ etc. statements and characterising these as either ‘loop’ or ‘case’ elements within the transaction. The model of such tool although being part of the characterization environment, its implementation is beyond the purpose of this report.

Method invocation opcodes are then characterised as to evaluate other characterised methods within the transaction. The bytecode that comprises the computation within and surrounding these elements are collated as bytecode blocks.

Listing 1 and Figure 1 depicts the Java BubbleSort implementation and its revised characterisation.

```
public void sort() {  
    for (int i = 0; i < a.length - 1; i++)  
        for (int j = a.length - 2; j >= i; j--)  
            if (a[j] > a[j+1])  
                swap(j, (j + 1));  
}
```

Listing 21 A Java Bubblesort Implementation.

Listing 2 is an implementation in Java of a bubblesort kernel. When compiled using the Characterisation parser implemented as part of the evaluation engine of JPACE, the bytecode produced is shown on the left of Figure 1, and on the right is the resulting performance characterisation of the method after running the transaction characterisation tool.

<i>BubbleSort/sort()V Bytecode</i>	<i>BubbleSort/sort()V Characterisation</i>
<code>icont_0</code> <code>istore_1</code> <code>goto 30</code>	<code><jPACe:bytecodeBlocks class="BubbleSort" method= sort"</code> <code>descriptor=" ()V" type="characterised"></code> <code><jPACe:bytecodeBlock id="sort()V:1"></code>
<code>aload_0</code> <code>getfield BubbleSort/a [I</code> <code>arraylength</code> <code>iconst_2</code> <code>isub</code> <code>goto 31</code>	<code><jPACe:loop count="{NElem} - 1"></code> <code><jPACe:bytecodeBlock id="sort()V:2"/></code>
<code>aload_0</code> <code>iload_2</code> <code>iload</code> <code>aload_0</code> <code>getfield BubbleSort/a [I</code> <code>iload_2</code> <code>iconst_1</code> <code>iadd</code> <code>iload</code>	<code><jPACe:loop count=" ({NElem} - 2) / 2"></code> <code><jPACe:bytecodeBlock id="sort()V:3"/></code>
<code>if_icmple 11</code> <code>aload_0</code> <code>iload_2</code> <code>iload_2</code> <code>iconst_1</code> <code>iadd</code>	<code><jPACe:case></code> <code><jPACe:probValue value="0.5"></code> <code><jPACe:bytecodeBlock id="sort()V:4"/></code>
<code>invokespecial BubbleSort/swap(II)V</code>	<code><jPACe:callMethod class="BubbleSort"</code> <code>method="swap" descriptor" (II)V"></code>
	<code></jPACe:probValue></code> <code></jPACe:case></code>
<code>inc 2 -1</code> <code>iload_2</code> <code>iload_1</code> <code>if_icmpge -30</code>	<code><jPACe:bytecodeBlock id="sort()V:5"/></code> <code></jPACe:loop></code>
<code>inc 1 1</code> <code>iload_1</code> <code>aload_0</code> <code>getfield BubbleSort/a [I</code> <code>arraylength</code> <code>iconst_1</code> <code>isub</code> <code>if_icmplt -55</code>	<code><jPACe:bytecodeBlock id="sort()V:6"/></code> <code></jPACe:loop></code>
<code>return</code>	<code></jPACe:method></code>

Figure 8The definition of bytecode blocks from the Java bytecode of the compiled Bubblesort algorithm.

Again, it should be noted that unconditional branch opcodes ('goto' for example) *are* included in bytecode blocks. The bytecode executed after the branch is also included within the same block until a conditional branch or method invocation opcode is found. Therefore, the bytecode block 'sort()V:1' from Figure 1 is defined as the first three opcodes of the method, as well as the opcodes starting from 'iload_1' (the opcode jumped to by the 'goto' opcode, also part of bytecode block 'sort()V:6') until the condition branch opcode at the end of the method ('if_icmplt').

5.2 Benchmarking Bytecode Blocks

The kernel of the new evaluation engine from JPACE has been developed that parses an XML-based performance model and calculates a predicted execution graph from the model's transactions and their relation to each other as described in the transaction map. The predicted response time obtained from a model's evaluation is the culmination of the all the bytecode block timings multiplied to the number of times they were executed during the course of the application's execution.

5.3 Implementation of benchmark toolkit (revised edition)

The benchmarking toolkit has been revised and re-developed to automate the process of benchmarking bytecode blocks on a given resource. A specific bytecode block is executed once, then twice, and so forth up to a total of 5000 iterations.

5.4 A theoretical hypothesis

Although research has been carried out on Hot Spot optimisation and this new implementation has been evolved to adapt to this technology, there is still considerable lack of information to pinpoint the relationship between the timing of a single bytecode or a sequential bytecode block and the number of iteration the benchmark measures at. Originally, it has thought without the optimisation this relationship would be linear.

i.e. duration of n times of bytecode (sequence or unit) = n x duration of one unit

However, according to the previous benchmark implementation, there were inconsistencies with the bytecode benchmark-timings against the number of iterations. **Each and every bytecode sequence or unit might associate with a function that varies according to the number of iterations.**

i.e. duration of n times of bytecode (sequence or unit) $y = f_y(n)$

f_y – parameterised function of the bytecode sequence or unit y .

This implies that the time to execute a bytecode unit is not proportional to the number of iteration it has been executed. Although this is based on assumption and historical data, it is a valid hypothesis to have been made when HotSpot optimisation is taken into account.

With this hypothesis, the aim of this revised benchmarking technique is to determine a pattern of Hot Spot optimisation to enable the benchmarking of bytecodes to be more accurate and hence provide better prediction on the JPACE framework.

5.5 Components of benchmark toolkit

As it was previously mentioned that instead of timing individual bytecodes, units of **sequential bytecode blocks (SBB)** were benchmarked. This led to a variation of both the toolkit written in Perl script and the prediction template file itself.

Due to this new concept, procedural sequence to carry out this benchmark process has been modified.

Several new scripts have been implemented for this new concept:

N.B. All scripts and directories are setup and run from

\$DIR = /dcs/00/csvee/private/research/work/cbs/ - variable \$DIR could be changed to the necessary directory.

xml_analyser.pl - extracts bytecodes from the characterisation xml into set of bytecode folders ready to be benchmarked.

./xml_analyser.pl <filename>

e.g. ./xml_analyser.pl fft

This command parses fft.xml at \$DIR/fft to output bytecode folders into fft/<bytecodeBlock>

As shown in previous chapters how characterisation files are constructed, below are some technical details of how to parse a revised version of the XML characterisation into an appropriate format for benchmarking:

An extract of the revised version of BubbleSort characterisation file

```
<jPACE:bytecodeBlocks class="uk.ac. ....BubbleSort">

  <jPACE:bytecodeBlock id="sort()V:1">
    <jPACE:OPCODE_istore_0/><jPACE:OPCODE_istore_2/>
    <jPACE:OPCODE_goto/><jPACE:OPCODE_istore_2/>
    <jPACE:OPCODE_aload_0/><jPACE:OPCODE_getfield/>
    <jPACE:OPCODE_arraylength/><jPACE:OPCODE_istore_1/>
    <jPACE:OPCODE_isub/><jPACE:OPCODE_if_icmplt/>
  </jPACE:bytecodeBlock>

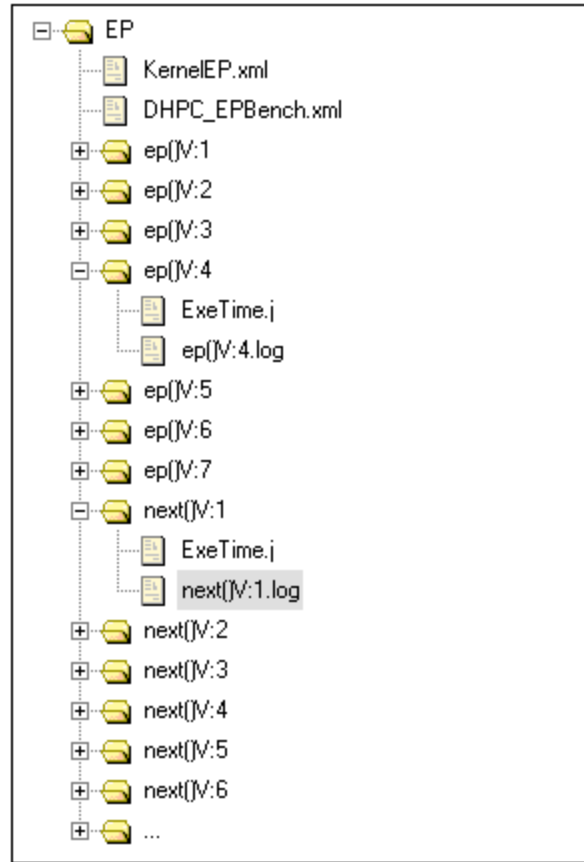
  <jPACE:bytecodeBlock id="sort()V:2">
    .
    .
  </jPACE:bytecodeBlock>
  .
  .
</jPACE:bytecodeBlocks>
```

Listing 22 shows an extract of the revised version of BubbleSort characterisation file

This is a typical SBB characterisation, it is important to note the characterisation of each method and object class shown; and to parse such block requires a formulated regular expression structures written in Perl script that captures each method characterisation with a XML file systematically. Below is the structure in pseudo-code:

```
##Detect the start of a method characterisation
$xml[$count] =~ m/<jPACE:bytecodeBlock\sid="/

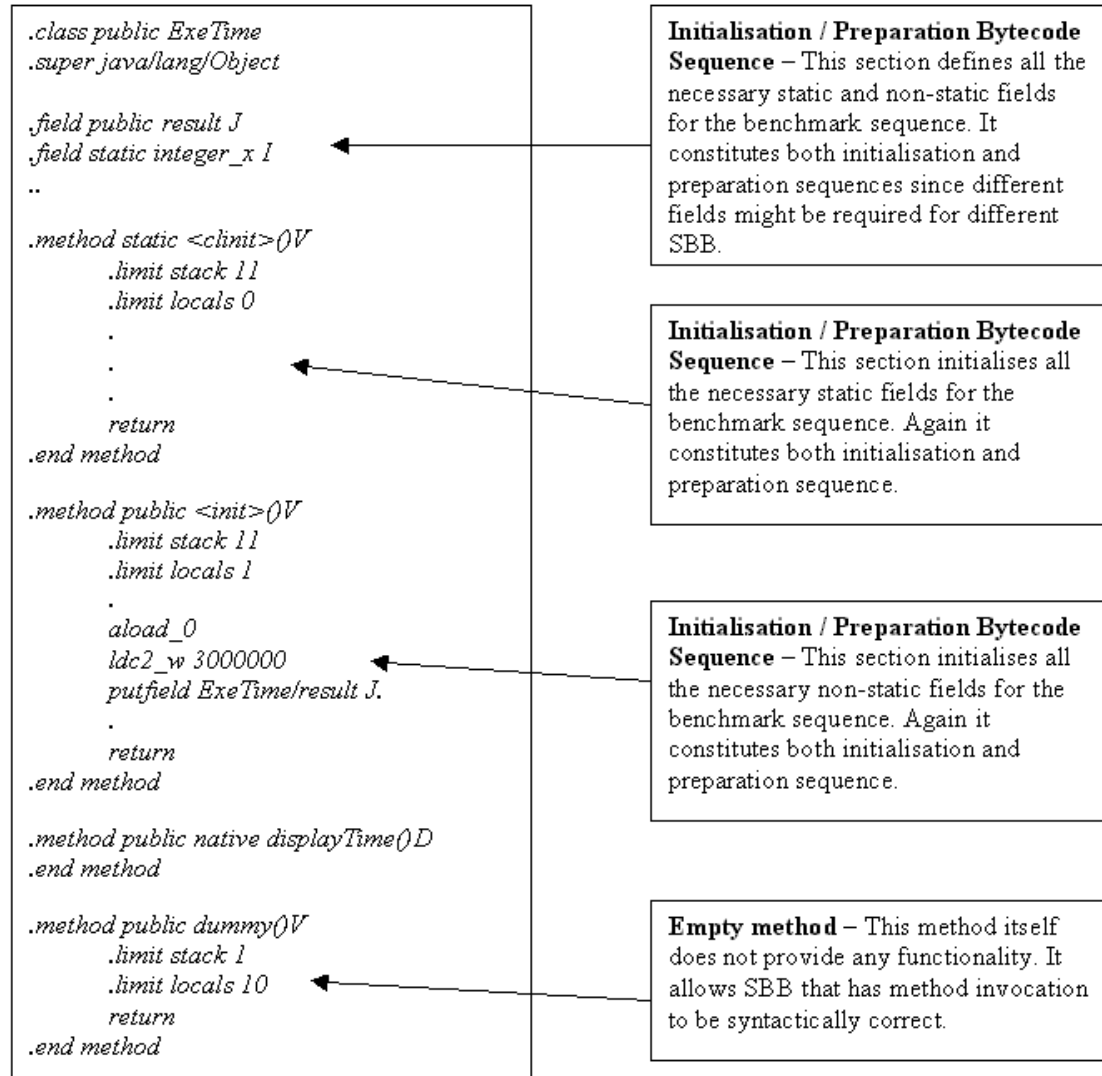
##To parse bytecodes (denoted by %%) from <jPACE:OPCODE_%%/> ##
$line =~ s/</>|\s//g;
@bytecode = split(/<jPACE:OPCODE_/, $line);
```

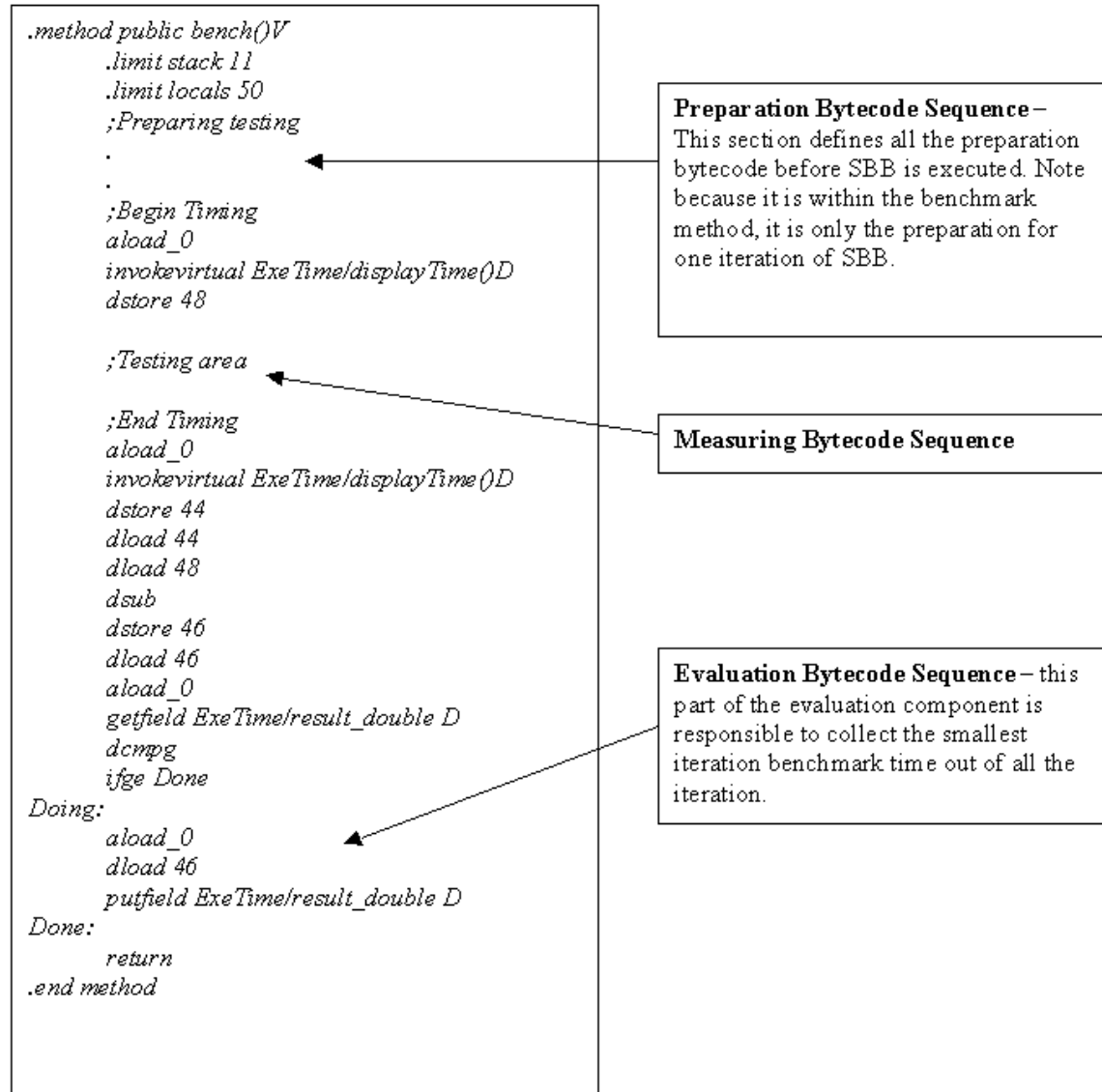
Screen 8 shows an example of repository of SSB for a Java application

This script parses a complete characterisation file into the Java application or **Java object characterisation repository** an example of which is shown in Screen 1. Within this repository, each Sequential Bytecode Block is parsed into its corresponding **SSB template directories** which consist of

1. **SSB template file** – this is the file similar to the **Template data file** defined at previous implementation, but instead of detailing measuring bytecode sequence and preparation bytecode sequence; it would now contain the unmodified SBB.
2. **Prediction template file** – this is the file **ExeTime.j** which is an object file that combines with the SBB template file completes the revised prediction template file. Below describes the technical detail of implementing this template file.



Listing 23 a section of the map of the template file written in Java bytecode.



Listing 24 a section of the map of the template file written in Java bytecode.

A copy of this file will be situated in each template directory. The next procedure is to prepare the prediction template file. Whereas previously the template data file would have contained all the relevant information and the script `./create_j.pl` will implement the prediction template file, the revised version requires a manual process and editing the prediction template file to include the SBB and its corresponding preparation sequence. Note it is beyond the time limit of this project to automate the process of carrying out the editing of the prediction template file. Below shows an example of editing the prediction template file. Although so far only BubbleSort.java has been mentioned, there are also

other Java applications that have been characterised and benchmarked, one of this application is a Java implementation of the **NAS Excessively Parallel benchmark**, which generates pseudo-random numbers with a Gaussian probability distribution. This benchmark is an integral part of the Kernel benchmark of the **DHPC Java Grande Benchmarks Suites (DHPC - Distributed and High-Performance Computing Group)**

SBB ep()V:5

```
dmul
dconst_1
dsub
dstore_1
ldc2_w
aload_0
invokevirtual
```

- SBB 5 of the method ep().
- Begins with dmul meaning there needs to be two type-double values on the top of the local stack.
- Set dummy method to take no argument.
- In the preparation bytecode sequence load two type-double values onto the stack.

Listing 25 SBB from the Excessively Parallel benchmarks characterisation

;Preparing testing

```
ldc2_w 1.0
ldc2_w 1.0
```

;Begin Timing

```
aload_0
invokevirtual ExeTime/displayTime()D
dstore 48
```

;Testing area

```
dmul
dconst_1
dsub
dstore_1
ldc2_w 1.0
aload_0
invokevirtual ExeTime/dummy()V
```

An extract of the prediction template file with the above SBB implemented into is shown on the left.

Listing 26 An extract of prediction templates with ep()V:5 SBB implemented into.

After all the prediction template files are prepared for benchmarking, the next phase is to gather results from these SBBs. A script has been implemented to automate this process.

cbs_iteration.pl – executes the prediction template files given a copy of the template file being situated in the same directory.

```
./cbs_iteration.pl <result_file>**
```

******A log file named <result_file> will be created with the benchmarked timing of the SBB in the prediction template file.

e.g. ./cbs_iteration.pl result.log

Since the first benchmark test is carried out on the Java Bubblesort algorithm implementation and the implementation only contains 7 SBB within the characterisation, therefore the script does not automate the whole process.

This script works on each SBB for the following numbers of iterations

From 0 – 100 at intervals of 5 iterations i.e. 5,10,15,20,...,95,100

From 100 – 5000 at intervals of 10 iterations i.e. 110,120,130,...,4980,4990,5000

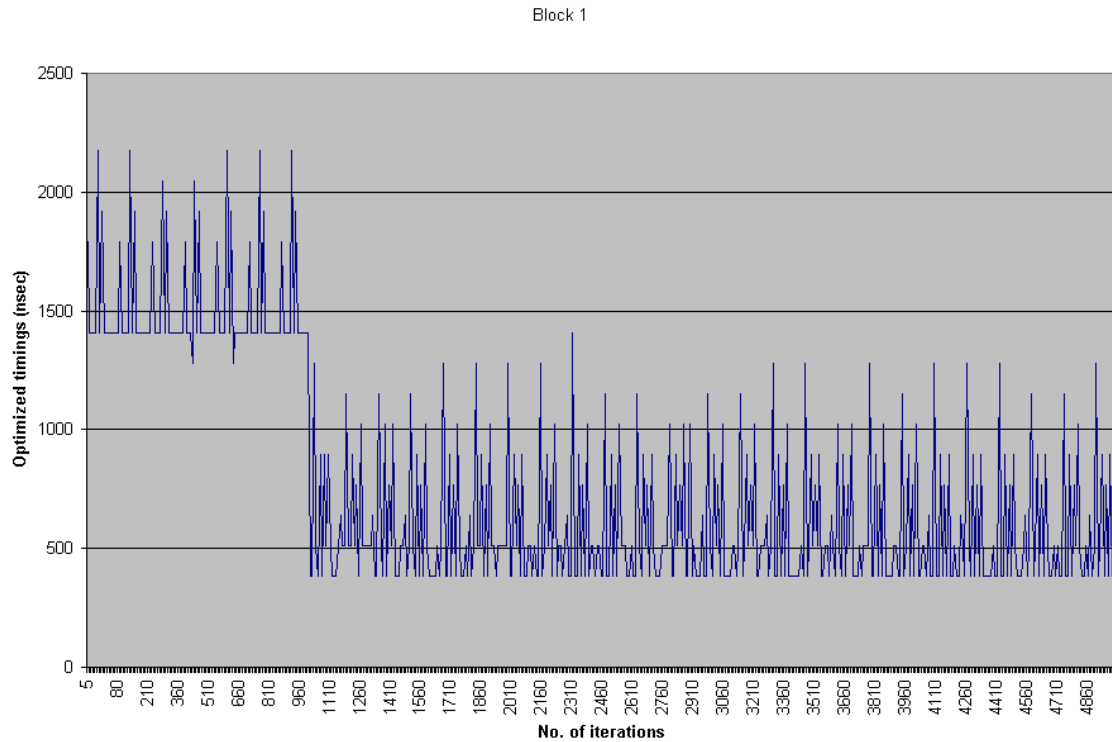
From 5000 onwards at intervals of 100 iterations i.e. 5100,5200,5300, ...

5.6 Preliminary Results and Understanding

Results of these benchmarked timings are gathered. Each log file, which contains the shortest running times of a sequential bytecode block collected from the some iterations ordered in ascending numerical order, is tabulated into graphical representation.

Figure 4 shows the results of such benchmarking technique on one of the sequential bytecode block from the Bubblesort method as defined by the automated transaction characterisation tool (graphical results of all seven blocks of SBB are attached to the appendix). It can be seen from the graph that there is a clear point (at roughly 1000 iterations) where the execution time of the bytecode block is significantly smaller than previously recorded, due to the fact that the hotspot compiler has chosen to optimise the block at this time. This value is used by the evaluation engine during a prediction in

choosing the appropriate response time for each bytecode block; if the bytecode block has been executed less than 1000 times so far during the course of the application then the higher average response time is used, otherwise it is the lower average response time.



Graph 1 The response time of bytecode block BubbleSort/sort()V:1 from 1-5000 iterations as the result of the benchmarking tool. It can be seen that at roughly 1000 iterations the block is optimised by the Hotspot compiler.

Up to this stage in development, by using the characterisation of the bubblesort algorithm shown earlier, and benchmarking the bytecode blocks on a given resource, the predicted and real execution time of the Bubblesort algorithm for varying data set sizes were obtained. Table 1 outlines these results. A percentage error of less than 30% is considered encouraging and a better understanding will help to ensure the elimination of this percentage error.

Size of Unsorted Array	Predicted Time (ms)	Measured Time (ms)	% Error
1000	160.82	156	3.1
10000	16021.11	12752	25.6
100000	1602275.98	1250869	28.1

Table 1 A comparison between the predicted execution time obtained from the evaluation engine and the actual measured execution time of the bubblesort algorithm for varying data sets

Throughout initial analysis of the result, there are also other interesting observations. One of them is the fluctuation of the results. These fluctuations may have been caused by background CPU overhead as the machines, which the benchmarks operate have included with other processes such as memory management and internal scheduling. In reality the application that has been benchmarked is usually run simultaneously with other processes within a high performance computational environment. Nevertheless these fluctuations although happen at a visually significant range, they conform to a consistent shape and this means that there are mathematical tools that can effectively reduce these fluctuations and since they are consistent it will not be detrimental to the accuracy of the results as earlier mentioned that the aim of this benchmarking session is to determine the patterns of optimisation and not the values. Since fluctuations are consistent and with the pattern of the graph. They suggest that there is an optimal value before optimisation begins and after optimisation begins.

i.e. no. of iteration $n \leq 1000$ then the running time of a bytecode unit is x
 no. of iteration $n > 1000$ then the running time of a bytecode unit is y

Where there is always x and y , which are the running time of a sequential bytecode block without and with Hot Spot optimisation respectively, associated with a particular SBB.

Therefore to refine the process of benchmarking, the next step is to identify these optimised and un-optimised running time. To compensate the fluctuation a statistical technique has been used. Since it is the average of the optimised and un-optimised values that are needed for performance prediction, the following technique is used:

To minimise this fluctuation error, it has been decided to exclude the xth iteration of which the running time is greater than 1 x standard deviation of the average running time for one iteration.

e.g. for 1000 iteration, the running time of a single SBB (T) is:

Suppose: $T = T_1 + T_2 + \dots + T_{1000} / 1000$ then...

$$T_{\text{optimised}} = T_1 + T_2 + \dots + T_x / x$$

where $T_1, T_2, \dots, T_x < T + \text{STDV}(T)$

Also according the structure of the bytecode prediction template, there are other parts of the benchmark file that could lead to consistent inaccuracy, below is a diagram to illustrate this problem:

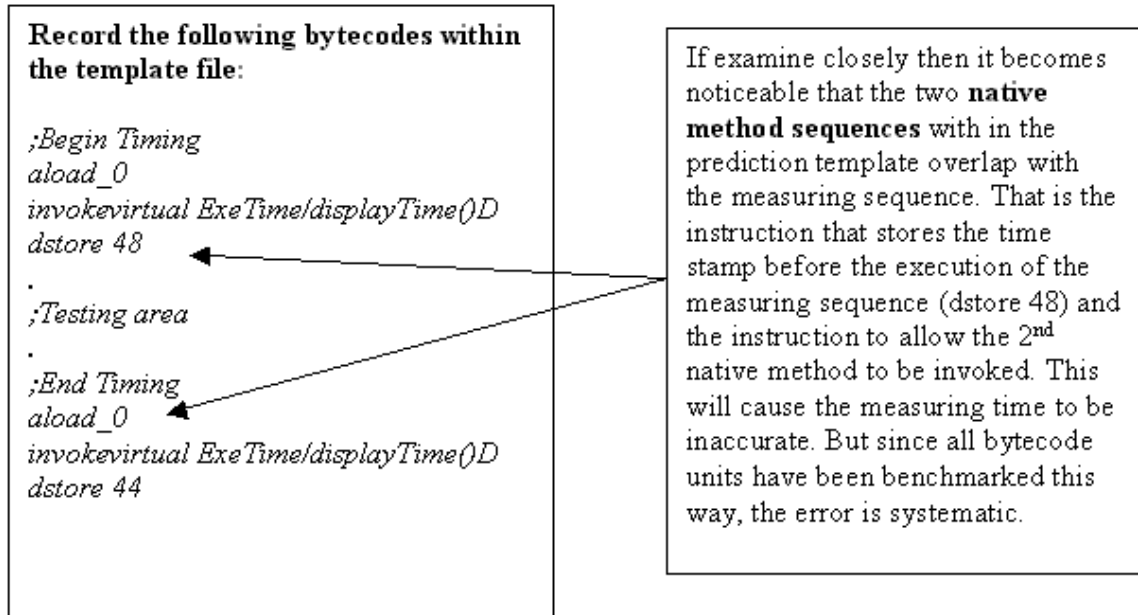


Figure 9 The inaccuracy of benchmarking caused by the overlapping of the template components

Therefore to ensure the timings that are obtained are solely the running time of these sequential bytecode blocks, each the average running time for one iteration of SBB is subtracted by the average running time for one iteration of no bytecode. By doing this each average timing is exactly the SBB running time.

5.7 High Performance Application (Benchmarks)

Also as mentioned earlier, the Java Bubblesort implementation is used as it is much more simplistic than other real-time high performance application and it is good as an indicator. Hence the following highly computational applications have also been considered [20]:

- ***NAS Excessively Parallel benchmark (EP)***
- ***2-D Fast Fourier Transform (FFT)***

*Both are part of the kernel section of the **Distributed and High Performance Computing Group (DHPC) Java Grande Benchmarks Suites***

To formally define, the aim of the next phase of benchmarking is to obtain a measure of the average unoptimised response time of the block, the average optimised response time, and the number of iterations at which the hotspot compiler decides to optimise the block. The results of each benchmark are stored in an XML-based resource object that is accessed by the evaluation engine during performance prediction.

Since these are realistic applications and their characterisations contain a lot more SBBs and they are more complex to be benchmarked. Therefore another set of toolkit has been developed to carry out all the procedure mentioned so far and the benchmark of all SBBs within an object characterisation will be carried out automatically. Furthermore, the toolkit will create five separate resources (five sessions) for all SBBs of both high performance application. These benchmarks have been carried out across different architectures. (depending on availability within the department, different machines which have the same hardware configuration are used to carry the benchmarking process)

Hostname: *budweiser.dcs.warwick.ac.uk*

Processor: *UltraSPARC®-III 360MHz*

Memory Size: *131072 KB*

Operating System: *SunOS 5.8*

JVM Version: *Java HotSpot™ Client VM (build 1.4.1) / (build 1.4.0)*

Hostname: *mscs.dcs.warwick.ac.uk*

Processor: *Intel® Pentium® 4 CPU 2.40GHz*

Memory Size: *531404 KB*

Operating System: *Linux kernel 2.4*

JVM Version: *Java HotSpot™ Client VM (build 1.4.1)*

Hostname: *labvista.dcs.warwick.ac.uk*

Processor: *Intel® Pentium® 3 i686 801.393 MHz*

Memory Size: *125244 KB*

Operating System: *Linux kernel 2.4*

JVM Version: *Java HotSpot™ Client VM (build 1.4.1)*

Preliminary result shows that due to different machines configurations and JVM versions, Hot Spot optimisation takes place at different iteration. With a more updated version JVM 1.4.1 the optimisation realises at 1000th iteration whereas for JVM 1.4.0 optimisation does not realise until 1500th iteration. Therefore the toolkit that carries out the renew-methodology of benchmarking SBB also caters for this issue.

5.8 Further refining the evaluation bytecode sequence

There are also re-development of the prediction template file to cater reduce the duration of benchmarking since such as FFT application contains over 70 sequential bytecode block within its characterisation. This means that trying to utilise the original method of benchmarking will be slow and inefficient.

To compensate for a more efficient benchmark, instead of comparing the benchmark time for every iteration within the method `bench()`, the benchmark time of every iteration will be outputted to standard output.

Below illustrates how the 3rd edition of the Benchmarking toolkit will use this further refined prediction template to benchmark high performance applications.

batch.pl - executes the prediction template files in the bytecodes folder in a pre-defined manner to obtain benchmark timings.

`./batch.pl <VM_version> <directory> <output_file>*`**

***A directory <directory>/dum is to be created for the script's temporary use.

e.g. `./batch.pl 1.4.1 fft fft/result/result.log`

This command executes the prediction template file in JVM 1.4.1, benchmark jasmin files in the folder fft/<bytecodeBlock> and output the timings to fft/result/result.log

Depending on what the <VM_version> is the following will show the most efficient strategy with the refined template file together with the toolkit implementation

Note: these are logic models that define the mechanics of the toolkits rather than the actual implementation

If it is JVM 1.4.1 then since optimisation realises at 1000th iteration the script will...

1. Executes template file (ExeTime.j) for iteration 1, ... ,1000 ten times

result_1000.log will contains 10000 un-optimised timing for that particular SBB

Note an argument 1000 means collect benchmark times for iteration 1 to 1000

```
for($i=0; $i<10; $i++) {  
    system("java ExeTime 1000 >> result_1000.log");  
}
```

2. Executes template file for iteration 1,...,6000 twice

result_6000.log will contains 10000 optimised timing for that particular SBB

Note an argument 6000 means collect benchmark times for iteration 1 to 6000

```
for($i=0; $i<2; $i++) {  
    system("java ExeTime 6000 >> result_6000.log");  
}
```

The toolkit only extracts timings from 1001...6000 iterations and since the benchmark is executed twice it also extracts from 7001 to 12000 since the second set of results is appended onto the same file. This is done by the following conditional statements.

```
if ( ($count > 1000) && ($count < 6001) ) || ($count > 7000) {  
    collect...  
}
```

This means in terms of benchmarking for optimisation and for non-optimisation, the averages are taken out of 10000 results.

A similar procedure is applied when benchmarking on JVM 1.4.0

For 1.4.0, the toolkit will still be collecting results for iteration 1,..., 6000. This occasion the script will run the template file for iteration 1,..., 1500 nine times. This means there will be 9000 un-optimised timings and by running the template file for iteration 1,..., 6000 and extracts only the timings for iteration 1501,...,6000 and for iteration 7500,...,1200 (two sets of results are appended onto the same ASCII file.) there will be 9000 optimised timings.

Next is to calculate the standard deviation of the data set (results)

Below is a Perl subroutine of the standard deviation implementation that the toolkit utilises:

```
sub std_dev() {  
  
    #The set of timings to be calculated  
    @array = @_;  
    #The total number of timings  
    $d = pop(@array);  
    #The total accumulation of all the timings  
    $total = pop(@array);  
  
    $var = 0;  
    foreach (@array) {  
        $var += ($_ - $total)*($_ - $total);  
    }  
  
    $var = $var / ($d - 1);  
    $std = sqrt $var;  
    return $std;  
}
```

Listing 27 a Perl subroutine for standard deviation

5.9 Negative valuation

When calculating the running time of each SBBs by taking away the running time of no-bytecode template file, there are instances, which result in computing negative values. Although it seems to be illogical theoretically, in practice since the resource of its computational environment is ever changing such as CPU cycle availabilities and memory caching mechanism being occupied by system processes running at background, fluctuations as seen previously occurs, therefore the standard deviation procedure is also used partially to eliminate this problem. Moreover, there are other measures taken to avoid negative valuation. These measures are the following:

- Only attempt to eliminate negative values if the sequential bytecode blocks consist of more than 4 bytecodes, this is because as the running time of a bytecode is comparatively short and although the time stamps were registered in nanoseconds, the resolution of the system clock is not small enough for the time stamp to be registered for accurate readings of the running time of less than four-bytecode units. Therefore by avoiding negative valuation will not have significant effect on the final result and moreover it will lead to inefficiency.
- If sequential bytecode blocks consist more than 4 bytecodes and negative valuation is realised, then attempt no more than 10 times in trying to obtain positive valuation as 10 attempts is the limit that has been decided to prevent detrimental effect on the performance of the benchmarking process.

Having discussed the technique and logical model for obtaining the optimised and un-optimised timings of each SBB, Screen 3 shows the content of one of the output file from the toolkit. Note the two distinct columns of un-optimised and optimised timing respectively, also the negative valuation at places of which SBBs consist less than 4 bytecodes

```
running 1.4.1
1 Standard deviation
no bytecode time before: 3381.376, 956.3392
no bytecode time after: 2504.52091903281, 925.103130939282
no bytecode std_dev before: 21083.6046302389, 2073.40570397227
no bytecode std_dev after: 612.825092171374, 783.181614050897
ep()V:1 before: 14620.5490828462, 1589.06629280993
ep()V:1 after: 12116.0281638134, 663.963161870652
ep()V:1 std_dev before: 22497.7626883237, 4934.76963771335
ep()V:1 std_dev after: 17403.4770392831, 830.959185748294
ep()V:10 before: 3424.85132014858, 1633.56526958087
ep()V:10 after: 920.330401115771, 708.462138641592
```

Screen 10 Default format of the <output_file>

Chapter 6

Result Evaluation

This chapter illustrates the experiment with the Java Grande Benchmark Suites and evaluates the comparison and scalability between the predicted execution times of those benchmarks using characterisation SBBs and their measured execution times.

The objective to implement these toolkits is to establish a framework for an efficient and accurate bytecode monitors. To ensure this framework meets its specification, below is a short description of a script that has been developed to parse SBB timings into a formatted repository so that it can be evaluated against evaluation engine's application prediction at real time.

```
batch_out.pl - outputs benchmarked timings from file output_file into a repository to be
parsed into the evaluation engine.

./batch_out.pl <VM_info> <machine_name> <characterisation_class>
<output_file>**

**A directory is to be created at the same directory level as batch_out.pl and set $STORE_DIR in the script to
correspond to this directory name

e.g. ./batch_out.pl x86.linux mscs-25 FFT fft/result/result.log

This script outputs benchmarked results from fft/result/result.log into appropriate format
JVM version - VM.sun.x86.linux-1.4.1
Machine - mscs-25.dcs.warwick.ac.uk
Characterisation_class - uk.ac.warwick.dcs.hpsg.applications.dhpc.fft.DHPC_FFTBench
```

Figure 1 is shows an example of the directory structure of the repository that will situate data to be collected and compared by JPACE's evaluation engine.

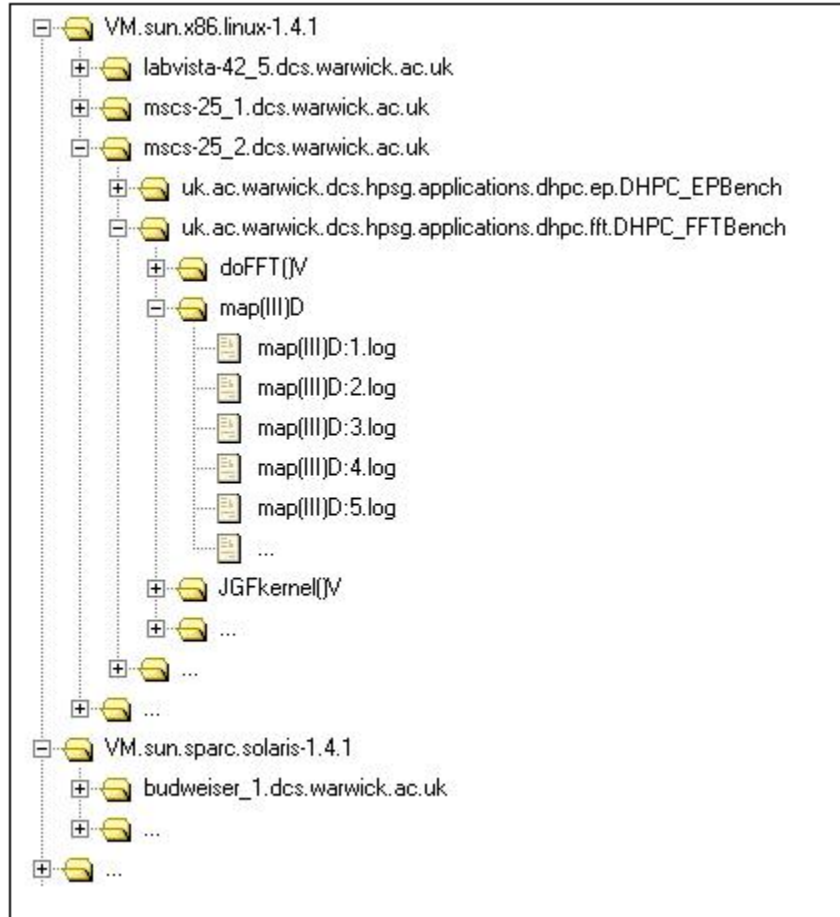


Figure 11 The directory structure of the result repository which resources timing are to be stored before being parsed by the evaluation engine.

The resource timings are then parsed into another XML characterisation file used by the evaluation engine. Figure 2 shows an extract of the XML file of which resource timings are parsed into.

```

<jPACE:classTiming class="uk.ac.warwick.dcs.hpsg.applications.dhpc.ep.DHPC_EPBench">

  <jPACE:methodTiming method="ep" descriptor="()V" noExecutions="0" avResponseTime="0.0">

    <jPACE:bytecodeBlockTiming id="ep()V:1">
      <jPACE:iterationTiming noIterations="1.0" executionTime="1197.88088888889"/>
      <jPACE:iterationTiming noIterations="1500.0" executionTime="1197.88088888889"/>
      <jPACE:iterationTiming noIterations="1501.0" executionTime="18.151822222223"/>
      <jPACE:iterationTiming noIterations="6000.0" executionTime="18.151822222223"/>
    </jPACE:bytecodeBlockTiming>
    ...
    ...
  </jPACE:methodTiming>

</jPACE:classTiming>

```

Figure 12 an extract of the XML characterisation file of which resource timings are parsed into.

Below are some results gathered by comparing between the predicted execution time obtained from the evaluation engine and the actual measured execution time of the FFT (Fast Fourier Transform) and EP (Excessively Parallel) benchmarks for varying data sets.

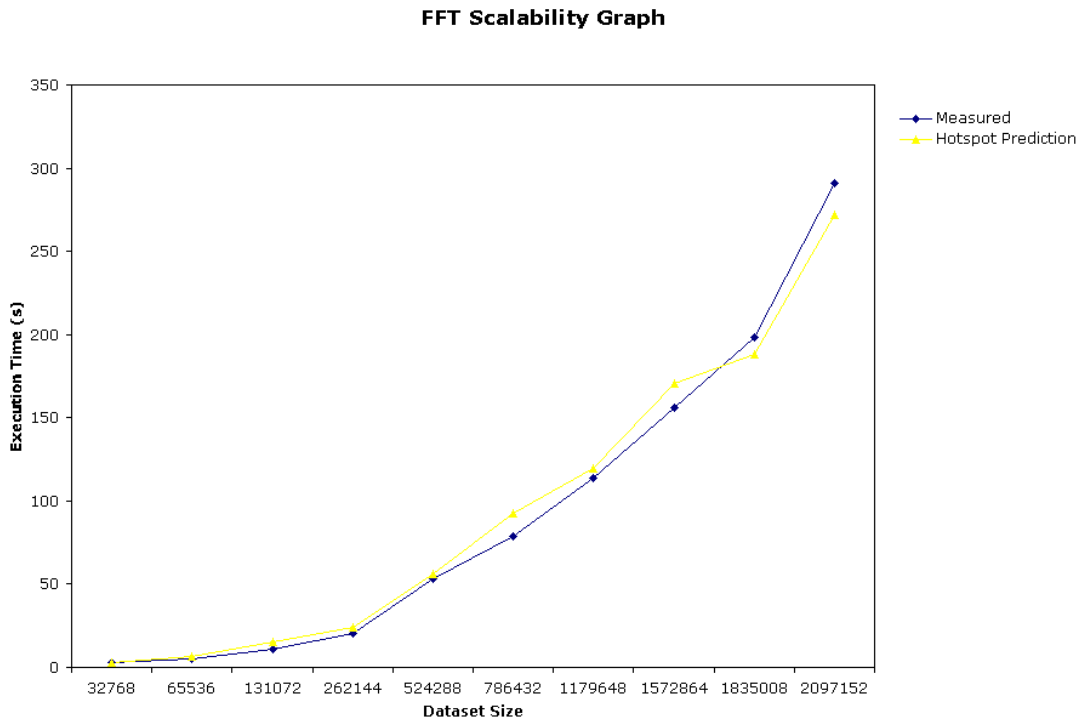
6.1 Fast Fourier Transform Benchmarks

Fast Fourier Transform – this benchmark performs a forward transform of a three-dimensional dataset. This kernel exercises complex arithmetic, shuffling, non-constant memory references and trigonometric functions. This is a CPU intensive benchmark working at the kernel level. It is commonly used in scientific computations which is a targeted area to utilise the Grid environment.

Each graph below corresponds to the table with the same number e.g. graph 1 corresponds to table 1.

Dataset (3D)	Hot Spot Predicted (seconds)	Measured time (seconds)	Percentage Error (%)
32X32X32	2.9140	2.5690	13.43
64X32X32	5.1650	5.1650	28.56
64X64X32	10.8380	10.8380	43.28
64X64X64	20.5060	20.5060	18.09
128X64X64	53.2104	53.2104	5.98
128X96X64	78.7750	78.7750	17.97
128X96X96	114.1094	114.1094	4.97
128X128X96	156.3070	156.3070	9.01
128X128X112	198.0904	198.0904	5.09
128X128X128	290.7728	290.7728	6.56

Table 2 shows the percentage error % of the predicted time against the actual running time on budweiser.dcs.warwick.ac.uk



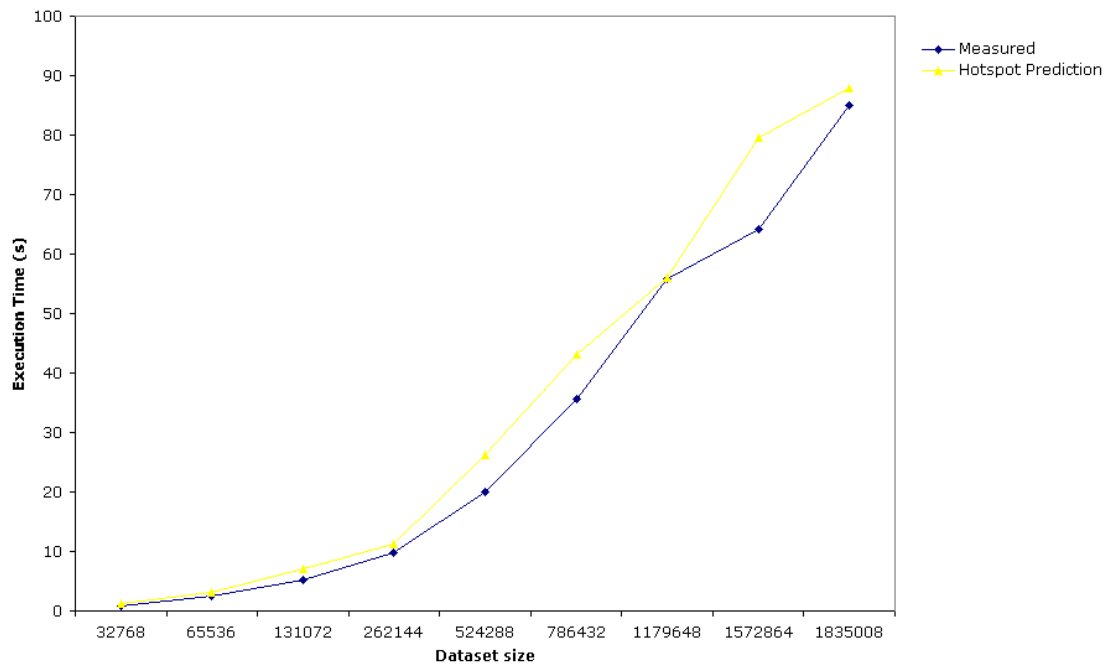
Graph 2 Predicted execution time and measured execution time comparison of the Fast Fourier Transform benchmark on machine budweiser.dcs.warwick.ac.uk

The sizes of the datasets are chosen to be powers of two as and this illustrates an exponential growth with the execution time, both predicted and measured. Although the sequential bytecode blocks are benchmarked to the nearest of nanoseconds, the execution time of the application that is characterised is more than one second and hence results are shown in seconds. The percentage in Table 1 shows all but one error is larger than 30% and this is encouraging especially when the datasets quite vary in size. In general if the average percentage errors from all the datasets are less than 30% then it can be classified as accurate since these predicted timings are used in conjunction with historic data within the performance characterisation environment of PACE. These historic data will certainly refine the accuracy. Moreover percentage errors that are more than 30% comes from the timings of datasets that are relatively small and when uncertainties in a dynamic computation environment arises that are independent to the size of the dataset then this uncertainty or error will be seen as significant. Fortunately the prediction technique that has been developed targets distributed applications in Grid environment and this suggests that the duration of these applications will allow this level of uncertainty that will be seen as insignificant. Table 2, which shows the percentage errors of the predicted time against the actual running time on labvista.dcs.warwick.ac.uk, again demonstrates the decrease of percentage error as the size of datasets increases and shows the insignificance of the uncertainty as the execution time increases.

Dataset (3D)	Hot Spot Predicted (seconds)	Measured time (seconds)	Percentage Error (%)
32X32X32	1.335	0.934	42.93
64X32X32	3.043	2.427	25.38
64X64X32	7.121	5.131	38.78
64X64X64	11.287	9.877	14.27
128X64X64	26.152	19.922	31.27
128X96X64	43.075	35.668	20.77
128X96X96	55.971	55.895	0.14
128X128X96	79.534	64.09	24.10
128X128X112	87.986	84.921	3.61

Table 3 shows the percentage error % of the predicted time against the actual running time on labvista.dcs.warwick.ac.uk

FFT Scalability Graph

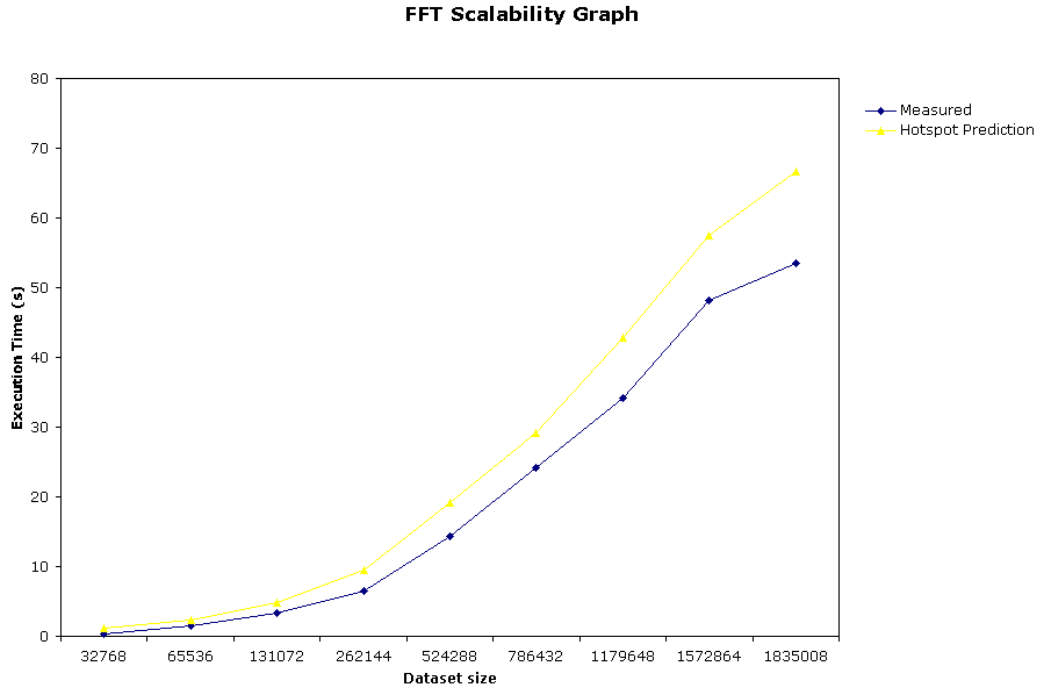


Graph 3 Predicted execution time and measured execution time comparison of the Fast Fourier Transform benchmark on machine labvista.dcs.warwick.ac.uk

In graph 2, which corresponds to the values in table 2, again suggests the proportionality of the predicted execution time against the measured execution time and the general decrease in the percentage error as the execution time increases.

Dataset (3D)	Hot Spot Predicted (seconds)	Measured time (seconds)	Percentage Error (%)
32X32X32	1.182	0.397	197.73
64X32X32	2.385	1.472	62.02
64X64X32	4.852	3.286	47.66
64X64X64	9.419	6.572	43.32
128X64X64	19.155	14.280	34.14
128X96X64	29.085	24.225	20.06
128X96X96	42.822	34.233	25.09
128X128X96	57.550	48.169	19.48
128X128X112	66.694	53.533	24.59

Table 4 shows the percentage error % of the predicted time against the actual running time on mscs.dcs.warwick.ac.uk



Graph 4 Predicted execution time and measured execution time comparison of the Fast Fourier Transform benchmark on machine mscs.dcs.warwick.ac.uk

Table 3 shows the percentage error % of the predicted time against the actual running time on mscs.dcs.warwick.ac.uk. In this table there is a percentage error, which is over 100% with a small dataset, at first this might seem to be as a sign of significant uncertainty. However this situation occurs when the predicted time is over 100% larger than the measured execution time and experiments show that a significant difference between the predicted time and the measured time only occurs when the execution time is small. As the machine hosted at mscs.dcs.warwick.ac.uk provides a much efficient computational environment in terms of processor's power and memory availability, in general the execution time of the FFT benchmark on this environment is smaller relative to the other machines hosted at labvista.dcs.warwick.ac.uk and budweiser.dcs.warwick.ac.uk, the percentage will generally be relatively larger but in terms of scalability, the prediction technique and its timing provided a good estimate across these hardware architecture and computational environments, as shown by the similarity in the curvature of the scalability graphs in graph 1, 2 and 3.

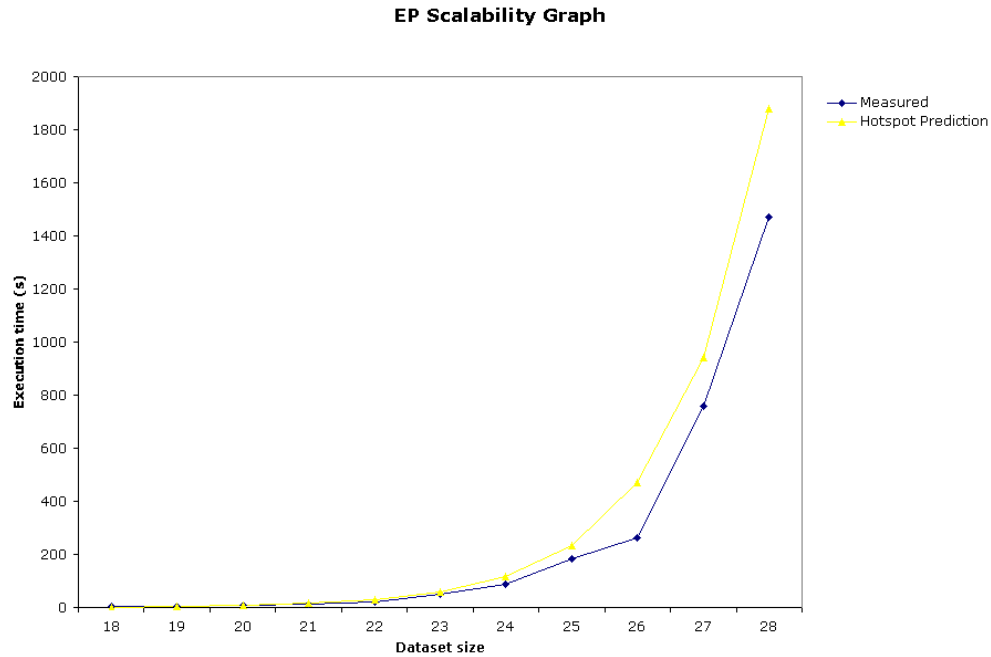
Clearly the pattern shown the benchmarked timings of the sequential bytecode blocks are accurate enough to allow the percentage error to drop well below 30%. A similar pattern can be observed for the comparison of the Excessively Parallel benchmarks shown in below.

6.2 Excessively Parallel Benchmarks

NAS Excessively Parallel benchmarks – This is one of the Java versions of the NAS (NASA Advanced Supercomputing Division) benchmarks, implemented by the DHPC (Distributed and High Performance Computing) Group. Its core function is to generate a pseudo-random numbers with a Gaussian probability distribution. The datasets are chosen and illustrated as powers of two for the convenience in terms of the binary operation within a computational environment. Graphs and tables are organised in a similar fashion to the FFT benchmarks experiments' results.

Dataset (power of 2)	Hot Spot Predicted (seconds)	Measured time (seconds)	Percentage Error (%)
18	1.859	2.091	11.10
19	3.694	3.038	21.59
20	7.365	6.984	5.46
21	14.706	11.316	29.96
22	29.329	22.318	31.41
23	58.755	49.520	18.65
24	111.487	89.373	31.46
25	234.950	183.929	27.74
26	469.877	260.700	80.24
27	939.730	759.574	23.71
28	1879.44	1471.370	27.73

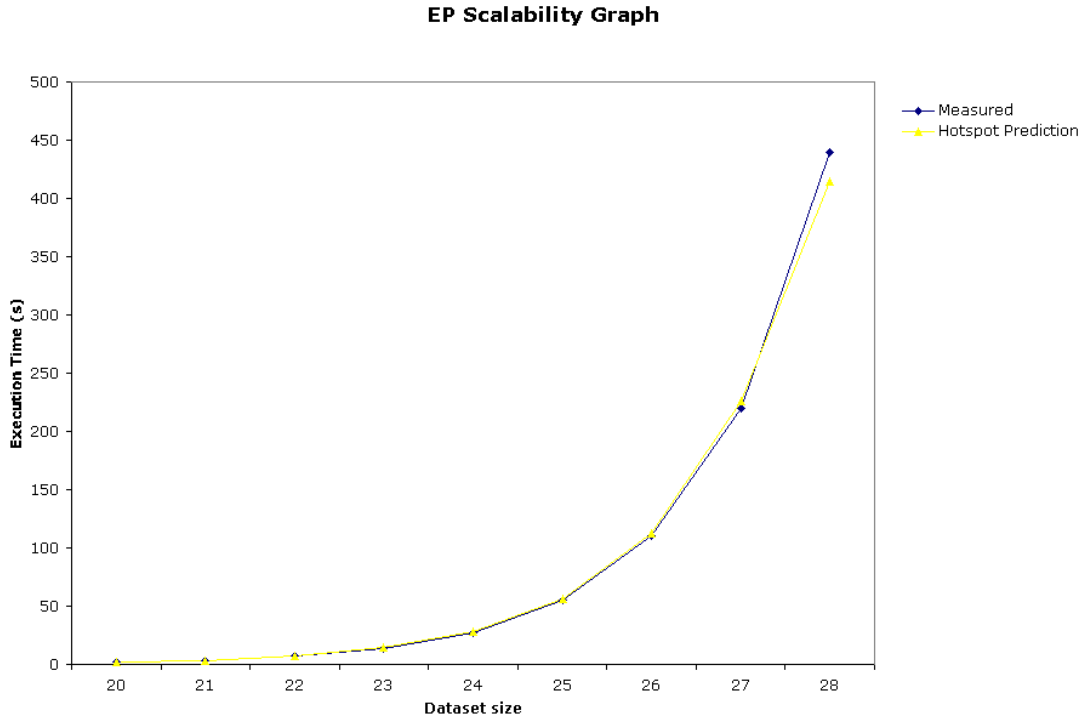
Table 5 shows the percentage error % of the predicted time against the actual running time on budweiser.dcs.warwick.ac.uk (Excessively Parallel)



Graph 5 Predicted execution time and measured execution time comparison of the Fast Fourier Transform benchmark on machine budweiser.dcs.warwick.ac.uk (Excessively Parallel)

Dataset (power of 2)	Hot Spot Predicted (seconds)	Measured time (seconds)	Percentage Error (%)
20	1.771	1.766	0.28
21	3.533	3.483	1.44
22	7.057	6.918	2.01
23	14.105	13.787	2.31
24	28.200	27.531	2.43
25	56.390	55.019	2.49
26	112.770	110.062	2.46
27	225.530	219.970	2.53
28	415.05	239.842	5.64

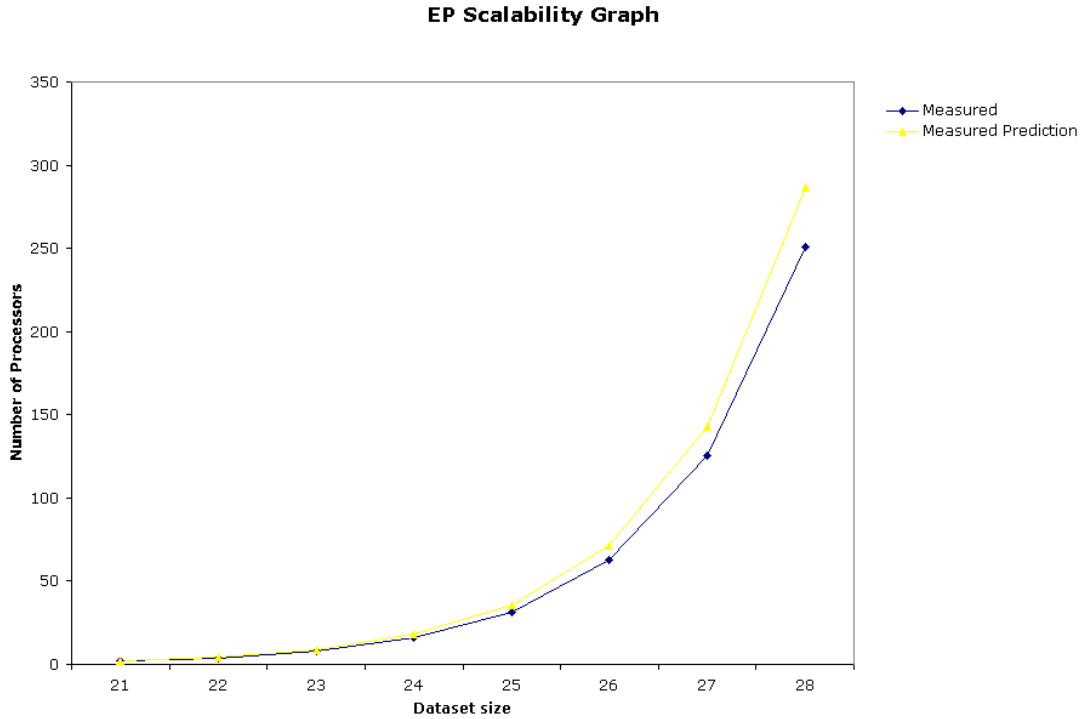
Table 6 shows the percentage error % of the predicted time against the actual running time on labvista.dcs.warwick.ac.uk (Excessively Parallel)



Graph 6 Predicted execution time and measured execution time comparison of the Fast Fourier Transform benchmark on machine labvista.dcs.warwick.ac.uk (Excessively Parallel)

Dataset (power of 2)	Hot Spot Predicted (seconds)	Measured time (seconds)	Percentage Error (%)
21	2.245	1.983	13.21
22	4.484	3.933	14.01
23	8.960	7.869	13.86
24	17.912	15.694	14.13
25	35.815	31.310	14.39
26	71.623	62.693	14.24
27	143.240	125.275	14.34
28	286.471	250.526	14.35

Table 7 shows the percentage error % of the predicted time against the actual running time on mscs.dcs.warwick.ac.uk (Excessively Parallel)



Graph 7 Predicted execution time and measured execution time comparison of the Fast Fourier Transform benchmark on machine mscs.dcs.warwick.ac.uk (Excessively Parallel)

The Excessively Parallel benchmark illustrates a different relationship to the Fast Fourier Transform benchmarks. This is mainly the case with the variation of the percentage errors even across different computational environments. By observing the graphical representation of the results (graph 4, 5, 6) it could be seen the relationship between the measured execution time and the predicted execution time collected at machine budweiser.dcs.warwick.ac.uk is more inconsistent in comparison with timings collected at machine labvista.dcs.warwick.ac.uk and mscs.dcs.warwick.ac.uk. This difference is clearly due to hardware configuration as the former machine has a different processing unit and has been installed with a different operating system to the latter two machines. Also since labvista.dcs.warwick.ac.uk and mscs.dcs.warwick.ac.uk offer a more efficient computational environment, to carry out experiments on small datasets would have given unrepresentative results and hence notably only larger datasets are examined with these two machines. Focusing on the latter two machines the percentage errors shown on table 5 and 6 are relatively small and consistent and this suggests a proportionality relationship

in between the uncertainty and the length of the execution time. In the experiments on labvista.dcs.warwick.ac.uk, a general percentage error of around 2 to 5% shown on table 5 also suggests the methodology that has incorporated the optimisation by Hot Spot technology is acceptable.

6.3 Other Benchmarks

To ensure the observations made are fair and accurate, other benchmarks within the Java Grande Benchmarks Suites have also been taken into consideration. These benchmarks operate across a collection of machines and this collection can be referred as cluster, which has also been the forefront topology for high performance computing and there experiments could suggest vital information for high performance Grid systems. These benchmarks are briefly described below:

- **Sparse Matrix Multiplication (SMM)** - This uses an unstructured sparse matrix stored in compressed-row format with a prescribed sparsity structure. This kernel exercises indirection addressing and non-regular memory references. A $N \times N$ sparse matrix is used for 200 iterations.
- **IDEA encryption algorithm benchmarks (Crypt)** - Crypt performs IDEA (International Data Encryption Algorithm) encryption and decryption on an array of N bytes. Performance units are bytes per second. Bit/byte operation intensive. [22]

Chapter 7

Conclusion

This chapter summarises the objective, specification and methodologies imposed in this project and evaluates the success, limitation and the future direction of this work.

7.1 Summary

With the emergence of grid computing, being able to intelligently allocate resource on a grid system to high performance computational application has been the key issue in high performance computing. To make resource allocations and high-level scheduling possible, management systems must be able to obtain characterisation information of these applications and this information must be supplied efficiently and with fair accuracy. One domain of the characterisation is the predicted application's execution time and as Java has become a popular programming medium.

This report has illustrated and discussed techniques of supplying predicted execution time by monitoring of Java bytecodes and also has documented the developmental stages of implementing this technique.

To conclude this report, the following is a summary of the developmental stages.

Through these developmental stages the following two methodologies were considered:

- **Timing analysis of Java bytecodes**
- **Method prediction on Java Programs**

The following is an overview of the methodology used for the Timing analysis of Java bytecodes:

1. Extracts individual bytecodes from programs

2. Collect the running time of repetitions of an individual bytecode, hence finding the average running time of a single bytecode.
3. Accumulate averages of all bytecodes to calculate the average predictive running time of the program

To efficiently implement this system, a systematic component-based framework has been developed, namely the **Bytecode Prediction Template**. It primarily consists of the following components:

- Initialisation bytecode sequence
- Preparation bytecode sequence
- Native method sequence (start time)
- Measuring bytecode sequence
- Native method sequence (end time)
- Evaluation bytecode sequence

Such a component-based template allows the benchmark process to be clearly described and it also means that further refinement is a lot more convenient. To utilise this template efficiently a collection of toolkits written in Perl has been developed. These toolkits has been explained in detail at the **first design and implementation** chapter. A Java Bubblesort algorithm kernel has been used to experiment the accuracy of this methodology.

The preliminary results show that...

The timing predicted by the first design and implementation did not exactly match the time measured from the BubbleSort.java 's performance critical section.

Observation suggested:

- Bytecode latency from invoking native method (calling C library).

- Speed difference between invoking same bytecodes that retrieve and assign values onto different variable.
- Effects of Just-in-Time compilation and Java Hot-Spot Optimisation.

Through the further development and understanding of Java optimisation a new methodology has developed which also utilises the Bytecode Prediction Template.

This new methodology employs a new bytecode unit called **Sequential Bytecode Block (SBB)** which can be defined as:

- Sequences of bytecodes that do not contain any conditional branch instruction or method invocation bytecodes.
- The size of each blocks is limited the size of a method permitted by the JVM specification.

Another collection of toolkits which is explained in the report have been developed to incorporate the functionalities of the Bytecode Prediction Template and SBB.

By monitoring the Bubblesort kernel characterisation SBBs, new observations were made and they were

- Depending on JVM and the resource the environment has, optimisation takes place at a specific iteration...
- Although individual benchmark session should be within a static resource, in reality resources such as CPU, memory changes at real time dynamically...
- Due to the ever-changing resource environment, fluctuations occur.

These observations have suggested the following modified hypothesis and formulation to cater these fluctuation and uncertainty:

To minimise this fluctuation error, it has been decided to exclude the xth iteration of which the running time is greater than 1x standard deviation of the average running time of one iteration.

To ensure this hypothesis is accurate and fair, several benchmarks from the Java Grande Benchmark Suite were experimented and notably the following two benchmarks have been studied in detail:

- **NAS Excessively Parallel benchmark (EP)**
- **3-D Fast Fourier Transform (FFT)**

Results by incorporating the new hypothesis are encouraging, although the relationship between the predicted and measured execution time varied across different computational environments (three different hardware-configured machines were used in this experiment), when the timings were tabulated onto some scalability graphs, the general curvature of the results suggests uncertainties due to fluctuation and hardware overhead were minimised and that the general percentage error between the predicted and the measured execution times is a lot lower than 30% and in some cases this error has been minimised to less than 1%. **These results confirm with sufficient confidence that a suitable bytecode monitoring technique has been devised for distributed Java applications within dynamic heterogeneous environments for the current release of Java Virtual Machine and Java Hot Spot optimisation technology.**

7.2 Limitation and Future Improvement

The following points detail the limitation of the Java bytecode monitor specified and described in this report. It should be noted that these points were not attended throughout the duration of project period due to the limit of time availability, technical knowledge and resources availability.

- **Effects of Just-In-Time compilation** – the main issue of characterising Java methods into sequential bytecode blocks is that it takes bytecode block out of the role of Just-In-Time (JIT) compilation, as the granularity of this technology is at least at the method level. This means inaccuracy might occur if JIT was utilised during the execution of any high performance Java application. This technology also encourages method in-lining (this is also the case with Hot Spot compilation), this leads the individual methods being effectively “merged” together and the course of this process may result in some SBB being removed and the characterisation model in this case will not bear the true characterisation.
- **Automate the modification of prediction template for all Java bytecodes** – due to the time availability the implementation to automate the insertion measuring bytecode sequence into the template file was not possible. This means that even though the theoretical base of this monitor is correct, in practice to enable the toolkit to work on real life high performance applications such as ones used in an e-business or an e-science environment will still not be ideal as these applications are relatively much larger and their characterisations will contain a lot more sequential bytecode blocks than the benchmarks of which this project’s bytecode monitor is tested on. To manually implementing each SBB onto the template files is not an efficient procedure for carrying bytecode prediction in a PACE environment, as predictive data should be available as quickly as possible.
- **Continuous development in Java Optimisation** – although the implemented bytecode monitor caters the Hot Spot technology, it is static in terms of other

optimisation such as with Just-In-Time compilation and this is due to the fact that these optimisation techniques and environments change very regularly. Such as the condition of Hot Spot detection and the heuristic for in lining are all subject to change and re-evaluation. For this characterisation needs to be a lot more dynamic and generically defined as with constant updating with virtual machines version means dramatic change in the mechanics of their optimisation and this can severely affect the accuracy of predicting application running time via bytecode monitors. Moreover these changes might focus a lot at a higher level granularity, mainly method level

7.3 Future Direction

As the results of these limitations, further research is highly recommended and areas of further work include:

- **Semantic definition** - The theory of meta-programming suggests a more dynamic approach is needed at the middleware level. A more semantically as well as syntactically defined intermediary /definition language can be evolved at this middleware level for high performance applications, especially in the performance-prediction domain. The motivation behind this idea came about from the area of Workflow management and its language for Workflow process definition. By evolving a middleware language focusing on a performance prediction domain, we can organise prediction entities in a more precise and meaningful way at a conceptual level, eg. the notions of subtasks and transactions can be defined as semantic entities but at the same time being able to be utilised at conceptual levels for more investigative work. Furthermore the introduction of a semantic implementation of such a language meant that future prediction work could be included with other metrics that relate to quality of services or workflow management, for example. This will not only allow a much more convenient way of designing and developing performance toolkit, but will also reinforce the ethos of GRID computing.

- **Automation and accuracy** - By refining the method of ARM (Application Response Measurement), the notion of a transaction mapping can be made more dynamic, and have the functionality of not just benchmarking sequential code but also of benchmarking multiples of applications based on other important metrics such as quality of service and workflow algorithms. Furthermore, if accuracy and automation can be enforced, then there is potentially a chance to look at the speed and transparency of prediction.

Reference

- [1] J. Meyer and Troy Downing, Java Virtual Machine, 1997
- [2] J. Engel, Programming for Java™ Virtual Machine, 1999
- [3] M. Schilli, Perl Power – A JumpStart Guide to Programming with Perl5, 1999
- [4] C. Herder and J. J. Dujmovic, Frequency Analysis and Timing of Java Bytecodes, pages 20-27
- [5] J.D. Turner, P.Y. Wong, S.A. Jarvis, A Brief Overview of the Performance Characterisation and Prediction of Java Applications
- [6] S.A. Jarvis, D.P. Spooner, H.N. Lim Choi Keung, G.R. Nudd, Performance Prediction and its use in Parallel and Distributed Computing Systems
- [7] G. R. Nudd, D. J. Kerbyson, E. Papaefstathiou, S. C. Perry, J. S. Harper, and D. V. Wilcox, PACE – A Toolset for the Performance Prediction of Parallel and Distributed Systems.
- [8] J. D. Turner, D. P. Spooner, J. Cao, S. A. Jarvis, D. N. Dillenberger and G. R. Nudd, A Transaction Definition Language for Java Application Response Measurement, pages 5-6
- [9] E. Papaefstathiou, D.J. Kerbyson, G.R. Nudd, T.J. Atherton, An Overview of the CHIP³S Performance Prediction Toolset for Parallel Systems, 8th ISCA Int. Conf on Parallel and Distributed Computing Systems, Florida USA. 527–533 (1995).

- [10] W. McColl, Foundations of time-critical computing. 15th IFIP World Computer Congress, Vienna and Budapest, 1998.
- [11] I. Foster and C. Kesselman, The GRID: Blueprint for a New Computing Infrastructure. Morgan-Kaufmann, 1998.
- [12] W. Leinberger and V. Kumar, Information power grid : The new frontier in parallel computing? IEEE Concurrency, 7(4), 1999.
- [13] P. Dinda, Online prediction of the running time of tasks. Cluster Computing, 5(3):225–236, 2002. Computing Systems (IPDCS98), pages 104–111,
- [14] High Performance Systems Group, University of Warwick
<http://www.dcs.warwick.ac.uk/~hpsg/>
- [15] T.Spencer, Benchmarking on HotSpot™, Hewlett Packard Corporation
http://h21007.www2.hp.com/dspp/tech/tech_TechDocumentDetailPage_IDX/1,1701,2022,00.html
- [16] J. Hardwick. Java Micro benchmarks
<http://www.cs.cmu.edu/~jch/java/benchmarks.html>
- [17] Java™ 2 Platform, Standard Edition, v 1.4.1 API Specification
<http://java.sun.com/j2se/1.4.1/docs/api/>
- [18] Java Native Interface
<http://java.sun.com/docs/books/tutorial/native1.1/index.html>
- [19] Java HotSpot™ Virtual Machine, Technical White Paper
http://java.sun.com/products/hotspot/docs/whitepaper/Java_HotSpot_WP_Final_430_01.html

- [20] DHPC Java Grande Benchmarking
<http://www.dhpc.adelaide.edu.au/projects/javagrande/benchmarks/>

- [21] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani, Overview of the IBM Java™ Just-in-Time Compiler
<http://www.research.ibm.com/journal/sj/391/suganuma.html>

- [22] EPCC Java Grande Benchmarking
http://www.epcc.ed.ac.uk/javagrande/index_1.html