A HARDWARE JAVA VIRTUAL MACHINE FOR HARD REAL TIME SYSTEMS

A thesis submitted to The University of Manchester for the degree of

Master of Philosophy

in the Faculty of Engineering and Physical Sciences

2005

GLENN COATES

SCHOOL OF ELECTRICAL AND ELECTRONIC ENGINEERING

List of Contents

Abstract 9
Declaration 10
Copyright Statement 11
Acknowledgements 12
1 Introduction 13
1.1 Introduction 13
1.2 Characteristics of RT software 14
1.3 Problems with RT Software Development 15
1.4 Problems with Java for RT 16
1.5 Temporal Issues for RT Software Development 17
1.5.1 Determination of Execution Time 20
1.6 Sources of Non-determinism in Real Time Systems 21
1.6.1 Programming Language and Usage 23
1.6.2 Operating System 25
1.6.3 Hardware 26
1.7 Implications for HRTSs 28
1.8 Summary 30
2 Using Java for Hard Real-Time Systems 32
2.1 Introduction 32
2.2 HRT Support Provided by Contemporary Languages 32
2.3 Benefits of using Java for General Software Development 35
2.4 Problems with using Java for Hard Real-Time 36
2.5 Real-Time Specification for Java (RTSJ) 39
2.6 Puschner, Bernat and Wellings HRT Profile 41
2.7 Critique of the Puschner Profile 42
2.8 Research Objectives 43
2.9 Summary 44
3 JVM Implementation Approaches 45
3.1 Introduction 45

3.2 Approaches 45
3.2.1 Interpretation 45
3.2.2 Just In Time Compilation 46
3.2.3 Ahead of Time Compilation 46
3.2.4 Hardware Acceleration 46
3.2.5 Hardware JVM 47
3.2.6 Choice of Hardware JVM 54
3.3 Work Required 55
3.3.1 Preliminary Work 55
3.3.2 Additional Work 56
3.4 Summary 58
4 The Development of JakHarta: a JVM to Support HRT Systems 60
4.1 Introduction 60
4.2 Development Environment 60
4.3 Support for Procedural Programming 60
4.3.1 The Java Virtual Machine 63
4.3.2 Behaviour of the invokestatic return and ireturn Bytecodes 65
4.3.3 Behaviour of the getstatic and putstatic Bytecodes 66
4.4 Constant Pool Resolution 66
4.5 JakHarta Tool Chain 67
4.5.1 Java Assembler 68
4.5.2 Java Compiler 68
4.5.3 WinZip 69
4.5.4 Romizer 69
4.6 Run-time Constant Pool for JakHarta 71
4.7 New Instruction Design 74
4.7.1 invokestatic 74
4.7.2 return and ireturn 77
4.7.3 getstatic and putstatic 78
4.8 Implementation of New Instructions 78
4.9 Summary 81
5 Test Results 83

5.1 Introduction 83
5.2 JBC Unit Tests..83
5.3 Simple Example 85
5.4 Complex Example 98
5.5 Summary 103
6 Conclusions and Future Work..104
6.1 Thesis Conclusions..104
6.2 Future work 106
Appendix 1 - Bytecode Execution Cycles..117
References 123

List Of Figures

FIGURE 1 EXAMPLES OF WORST CASE EXECUTION TIMES. 22

FIGURE 2 INPUT DATA DEPENDENCY. 23

FIGURE 3 PICOJAVA BLOCK DIAGRAM (TAKEN FROM [SUN96]). 48

FIGURE 4 LAVACORE BLOCK DIAGRAM, TAKEN FROM [DER01]. 50

FIGURE 5 MOON2 JAVA PROCESSOR, TAKEN FROM [VUL03]. 51

FIGURE 6 JEM2 ARCHITECTURE, TAKEN FROM [HAR]. 53

FIGURE 7 PROCEDURAL JAVA PROGRAM USING STATIC METHODS AND VARIABLES. 61

FIGURE 8 JAVA ASSEMBLY BYTECODE FOR THE DEMO AND CALC METHODS. 62

FIGURE 9 JVM STACK SHOWING 2 ACTIVATION RECORDS. 64

FIGURE 10 JAKHARTATOOL CHAIN DATA FLOW DIAGRAM. 67

FIGURE 11 ROMIZER DATA FLOW. 70

FIGURE 12 JAKHARTA RUNTIME CONSTANT POOL AND MEMORY LAYOUT. 72

FIGURE 13 DATA AND RETURN STACKS FOR A METHOD INVOCATION. 76

FIGURE 14 DATA AND RETURN STACKS AFTER A RETURN/IRETURN. 77

FIGURE 15 UNIT TEST FOR IFEQ BYTECODE. 84

FIGURE 16 PROCEDURAL JAVA PROGRAM USING STATIC METHODS AND VARIABLES. 85

FIGURE 17 MEMORY LAYOUT OF SIMPLE JAVA PROGRAM. 86

FIGURE 18 *MAIN* METHOD BEFORE CALLING *DEMO*. 87

FIGURE 19 AFTER CALLING *DEMO*. 88

FIGURE 20 BEFORE CALLING THE *CALC* METHOD. 90

FIGURE 21 AFTER THE *INVOKESTATIC* FOR *CALC*. 91

FIGURE 22 *CALC* METHOD PRIOR TO THE *GETSTATIC*. 92

FIGURE 23 AFTER *GETSTATIC* IN *CALC*. 93

FIGURE 24 *CALC* METHOD PRIOR TO *PUTSTATIC*. 94

FIGURE 25 AFTER THE *PUTSTATIC* IN *CALC*. 95

FIGURE 26 BEFORE THE *RETURN* IN *CALC*. 96

FIGURE 27 AFTER THE *RETURN* IN *CALC*. 97

FIGURE 28 RECURSIVE PROGRAM WHICH CALCULATES THE FACTORIAL OF A GIVEN INTEGER. 99

FIGURE 29 DAG FOR RECURSIVE FACTORIAL PROGRAM. 100

FIGURE 30 PRIOR TO THE FIRST INVOKESTATIC FOR *FACTORIAL*. 102
FIGURE 31 AFTER THE FINAL RETURN FROM *FACTORIAL*. 102
FIGURE 32 CLASS DIAGRAM SHOWING DETERMINISTIC IMPLEMENTATION OF METHOD TABLES. 110
FIGURE 33 SINGLE METHOD TABLE FOR A CLASS HIERARCHY. 111
FIGURE 34 NON-DETERMINISTIC INTERFACE METHOD TABLES. 112
FIGURE 35 DETERMINISTIC INTERFACE METHOD TABLES. 113
FIGURE 36 DETERMINISTIC INTERFACE METHOD TABLES. 115

List Of Tables

TABLE 1 FEATURE PRIORITISATION. 56
TABLE 2 NEW JAKHARTA INSTRUCTIONS ADDED TO JASMIN ASSEMBLER. 68
TABLE 3 CONSTANT CLOCK CYCLE TIMES FOR NEW INSTRUCTIONS. 81
TABLE 4 SUMMARY OF CYCLE TIMES FOR NEW INSTRUCTIONS. 81
TABLE 5 BYTECODE CYCLE TIMES FOR BASE JVM AND JAKHARTA. 122

List of Expressions

EXPRESSION 1 WCET CALCULATION FOR FACTORIAL METHOD. 101
EXPRESSION 2 SIMPLIFIED WCET CALCULATION FOR FACTORIAL METHOD. 101

List of Acronyms

ALU	Arithmetic Logic Unit
BCET	Best Case Execution Time
CP	Constant Pool
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DMA	Direct Memory Access
DS	Data Stack
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
HRT	Hard Real Time
HRTS	Hard Real Time System
IR	Instruction Register
ISR	Interrupt Service Routine
JAR	Java Archive
JBC	Java Byte Code
JIT	Just In Time
JNI	Java Native Interface
JTAG	Joint Test Action Group
JVM	Java Virtual Machine
LIFO	Last In First Out
OO	Object Oriented
OS	Operating System
PC	Program Counter
QOS	Quality Of Service
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RMA	Rate Monotonic Analysis
ROM	Read Only Memory
RS	Return Stack

RSP	Return Stack Pointer
RT	Real Time
RTCE	Real Time Core Extensions
RTDS	Run Time Data Structures
RTJEG	The Real Time Java Expert Group
RTL	Register Transfer Language
RTOS	Real Time Operating System
RTS	Real Time System
RTSJ	Real Time Specification for Java
SB	Stack Base
SP	Stack Pointer
SRT	Soft Real Time
SRTS	Soft Real Time System
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
WCAO	Worst Case Administrative Overhead
WCET	Worst Case Execution Time

Abstract

Hard real-time systems have strict behavioural and temporal requirements, which must be met under all runtime situations. Hard real-time applications must typically be programmed at a low level, which is a time consuming and error prone task. In recent years, there has been significant interest in raising the level of abstraction at which hard real-time systems are programmed. Java has been suggested as a candidate programming language since it provides support for features that are important for many real-time applications.

This thesis is concerned with investigating the practical needs of hard real-time systems whilst assessing the suitability of Java for such environments. The work defines and partly implements a hardware based Java machine and elements of a tool chain based on the findings.

Declaration

No Portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Copyright Statement

Copyright in text of this thesis rests with the Author. Copies (by any process) either in full, or of extracts, may be made **only** in accordance with instructions given by the Author and lodged in the John Rylands University Library of Manchester. Details may be obtained from the Librarian. This page must form part of any such copies made. Further copies (by any process) of copies made in accordance with such instructions may not be made without the permission (in writing) of the Author.

The ownership of any intellectual property rights which may be described in this thesis is vested in The University of Manchester, subject to any prior agreement to the contrary, and may not be made available for use by third parties without the written permission of the University, which will prescribe the terms and conditions of any such agreement.

Further information on the conditions under which disclosures and exploitation may take place is available from the Head of School of Electrical and Electronic Engineering.

Acknowledgements

I would like to extend thanks and gratitude to a number of people:

- Dr Peter Green for supporting me as a part-time, home-based student. I am grateful for our conversations, arguments and your guidance during the past three years. I am also grateful for your help in writing the IEE paper [COA04], which was based on this work.

- Dr Kelvin Nilsen of Aonix Inc. for our practical discussions.

- Members of the University of York Safety Critical email group [YOR01], for responding to my questions.

- Celoxica Ltd, for allowing me to use the Handel-C DK2 development kit.

- Most of all, my wife Kathleen who has supported me during this work.

1 Introduction

1.1 Introduction

A Real-time system (RTS) is a computer system that typically monitors and/or controls some larger product or system. RTSs are ubiquitous and relied upon in many areas of modern life. They can be found in many diverse application areas such as avionics, military, telecommunications, automotive, process control, consumer devices and household appliances. Often the user sees the Real-time (RT) system as an electrical or mechanical device and, in such cases, they are also referred to as embedded. These systems may need to operate for many years in hostile environments and have the potential to cause great inconvenience or harm if they fail.

Many RTSs must interact with their environments on time scales that are dictated by the environment. Such systems have time constraints, or deadlines. Hard deadlines must be met for all runtime scenarios including peak loads and exceptional circumstances. There is no benefit in late delivery of the data. Hence, it is imperative that hard deadlines are met otherwise the application correctness is seriously compromised [BRI99], [KOP97], [BUR97]. For example, a car engine controller needs to calculate the amount of fuel and the exact moment at which the fuel must be injected into the combustion chamber of each cylinder. These calculations must be completed within the required deadlines otherwise engine damage is likely which could potentially lead to an accident.

Soft deadlines need to be met but some missed deadlines can be tolerated as long as not too many deadlines are missed and they are not missed by much time [KOP97]. The late delivery of the data or response is still of value. The system is unlikely to fail although the Quality of Service (QOS) is degraded. For example, missed deadlines in a video conferencing system may result in some display jitter, however it does not prevent the system being used effectively.

Characteristics of deadlines can be used to categorise RTSs. A system with only soft deadlines is said to be a soft real-time system (SRTS), whereas a system with one or more hard deadlines is said to be a hard real-time system (HRTS) [KOP97].

Many HRTSs are safety critical in that a failure such as a missed deadline may have catastrophic consequences such as loss of life or environmental damage. As such, a key development driver for many HRTSs includes safety certification [KOP97]. Hence, many HRTSs are validated using static analysis in order to ensure that they will behave predictably and will meet deadlines under all runtime scenarios, not just those observed by testing. This is because generally, testing can only show the presence, not the absence of bugs [DIJ69]. In contrast, the consequences of a missed deadline for a SRTS are far less severe. A less rigorous approach to development is usually taken where ad-hoc and empirical techniques are often used.

1.2 Characteristics of RT software

On the basis of the discussion of Section 1.1 it is possible to identify a number of generic requirements that must be met by RT software. These apply to both HRTSs and SRTSs.

A RT application domain is inherently concurrent which means it must monitor and control several different aspects of the environment simultaneously. For example, a fly-by-wire system needs to control a number of different control surfaces at the same time. It must also monitor the state of the control surfaces and various sensors to ensure that the plane is following the required flight path.

As indicated in Section 1.1, a RTS must respond to timescales, which are typically small compared with the timescale of the environment. RTSs are temporally predictable, which means that the time taken to produce an output or result is consistent within specified bounds.

RTSs exercise autonomous control and often operate unattended for long periods of time. As such they must be dependable. There are a number of aspects of dependability

including reliability, safety, availability and maintainability [LAP85]. Some RTSs (usually HRTSs) control aspects of the environment that can potentially cause harm to people. Such systems are safety critical. It is suggested that some safety-critical systems may need to limit the probability of failure to as low as
10^{-9} per operational hour [KOP97].

RTSs typically interact with their environment via sensors and actuators; therefore they must be able to interface with non-standard hardware. Consequently the range of peripheral devices that must be accommodated is broader than those typically found in desktop computer systems. Therefore the RTS must contain bespoke low-level drivers for such hardware devices.

1.3 Problems with RT Software Development

The typical characteristics of RTSs outlined in Section 1.2 mean that they are often programmed at a low level and therefore naturally platform specific. An intimate knowledge of the application-specific devices and other system components including the real-time operating system (RTOS) is required. Programming may be done using languages such as C or even assembler. The target system is often very different from the host development system and peripheral devices may not be immediately available at the start of software development. Debugging RTSs can be problematic due to their timing sensitive nature. This means that complex tools such as in-circuit emulators and logical analysers may be required in addition to traditional software debugging tools. As discussed in Section 1.1, some RTSs (usually HRTSs) need to validate behaviour and timing using static analysis.

The difficulty in developing RT software means that productivity is low. Mistakes can easily be made which can lead to faults with serious or even catastrophic consequences. The skill level and knowledge required for RT software is much higher and more specialised than for general application development. RT software can be 2 to10 times more expensive to develop and maintain than general-purpose application software [KOP97].

It would be beneficial if a higher-level approach could be applied to the development of RTSs. Modern programming languages provide a number of features and allow for improved abstraction, encapsulation, type safety and memory management. Java has been described as a simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language [SUNA][1]. Java is interesting because it is a high-level language, which has been successfully used for desktop and enterprise applications [REX97]. Java directly addresses some of the characteristics of RT software. For example, it supports threads and concurrency and reliability is supported via exceptions and automatic memory management. Java supports a simplified object-orientated (OO) model, and it has been suggested that this helps in developing well-structured, maintainable and reusable software. Java is widely taught at universities and used extensively in industry, which means there is a large pool of experienced programmers. Hence there is interest in using Java for RT in order to increase productivity and address the problems inherent with RT software development. This thesis is concerned with investigating the form of run-time support that should be provided if Java is to be used for HRTSs.

1.4 Problems with Java for RT

Although desirable from a software engineering point of view, there are serious practical difficulties in using Java for RTSs. This is especially true in the context of its use in HRTSs. A key issue is the execution model (which is discussed further in Sections 4.3.1 and 4.4). Java programs are typically compiled into an intermediate code, known as Java bytecode (JBC), which is then interpreted by a software virtual machine, the Java Virtual Machine (JVM). A JVM implementation is typically a large and very complex piece of software written in C, which interacts with the underlying operating system (OS). As a consequence many JVMs are slow and a Java based system often requires more memory when compared with an equivalent system developed in C. In general an interpreter based JVM will be many times slower than native code [MAN98]. A number of factors mean that temporal determinism is difficult to achieve, and these are discussed in Section 1.5. Because Java is mainly used in desktop and

[1] More correctly, there are a number of different things that go by the name Java. This includes a language, a virtual machine model, a large class library and a class-loading model.

enterprise domains the issue of temporal determinism has not been considered much. Added to problems relating to the Java execution model, there are a number of problems with the language itself. These will be discussed in Section 2.4.

As a consequence of the problems with Java for RTS development, there have been a number of proposals as to how Java can be modified to make it more suitable for use in RTSs in general, and HRTSs in particular. These proposals are reviewed in Sections 2.5 to 2.7. A feature of most proposed versions of Java for RTS is that they focus on adding features needed by RT applications, and removing those that are unhelpful. However, there has been little work reported in the literature concerning the type of run-time support needed for RT versions of Java. In other words, little consideration of the type of JVM needed to support RT Java. Hence the purpose of this thesis is to investigate JVMs for HRTSs with particular emphasis on temporal determinism, and to take some steps towards the development of such a JVM. For reasons that will be discussed in Section 3, the focus is on JVMs implemented in hardware.

1.5 Temporal Issues for RT Software Development

By definition, RTS are computer-based systems that must reliably and predictably perform time-constrained computations. Since timing issues are very important in HRTSs, temporal issues will now be briefly discussed.

Timing constraints typically relate to timing relationships between environmental stimuli and RTS responses to those stimuli. The timing relationships are usually dictated by the dynamics of the environment [KOP97]. A very common form of timing constraint is the deadline where, upon receiving a stimulus from the environment, a system must compute a response and deliver it back to the environment within a specified time interval or deadline.

A common implementation model for a RTS is as a set of concurrently executing processes or tasks, which typically interact in order to compute the response to environmental stimuli. Tasks may be periodic, in which case they are executed at a fixed rate. Periodic tasks are often used for functions such as sampling or monitoring.

Alternatively tasks may be aperiodic, in which case they are triggered unpredictably, sometimes by interrupts. In systems where timing is important, each task typically has its own deadline, which is the time between the task being triggered for execution, and it delivering its response to the environment.

In many RTSs there are fewer physical processors than there are tasks, in which case a set of tasks must share a processor. Consequently the issue of the *scheduling* of tasks becomes important to ensure that timing requirements are met. In RTSs the scheduling task is concerned with finding an order of task execution such that all tasks in the system meet their deadlines. Scheduling can be performed at design time, in which case it is called static or off-line scheduling. Alternatively it can be performed at runtime and called on-line scheduling.

Static scheduling typically produces a cyclic executive implementation based on a set of periodic tasks, and is the most common approach in HRTSs [XU93][2]. Cyclic executives are based on a time-triggered control loop, which, sequentially cycles-through and executes a set of tasks. This is highly predictable as the programmer has design time knowledge of when each task is scheduled. It is also popular due to its runtime simplicity, typically requiring a small (or no) RTOS. Because tasks are not pre-empted there is no need for mutual exclusion, which further reduces required RTOS features and simplifies the system. However there are a number of problems with cyclic executives. If there is a need for substantially differing deadlines, then the cyclic executive can become impractical. For example, tasks may need to be split into a number of smaller tasks and dispersed through the cyclic executive in order that a schedule can be built. This makes maintenance difficult as it may require re-writing of tasks and the cyclic executive loop.

If scheduling is to be performed at runtime, an RTOS or equivalent is required. A key element is the scheduler. The scheduler executes the system's scheduling policy, which determines which task should run next. Many scheduling policies have been proposed in

---

[2] There are techniques for integrating aperiodic tasks by bounding the rate at which they are received.

the OS literature. The most commonly used policy in RTS is the priority-based pre-emptive policy. Each task is given a fixed priority, which in general purpose systems is some measure of its perceived importance to the system, but in RTSs is typically related in an indirect fashion to its deadline. The scheduler ensures that the task that has the highest priority that is able to run is always the task that is run next. Hence the scheduler typically uses a queue of tasks that are ready to run (referred to as the ready queue) which is sorted in order of priority. When the scheduler is called upon to make a scheduling decision, it consults the ready queue and checks whether the task at the head of the queue has a higher priority than the task that is currently executing. If not, the currently executing task typically continues. If so, then the executing task is pre-empted which means that the state of its computation is saved. It is placed on the ready queue in a position determined by its priority, and the task at the head of the ready queue is executed.

Mathematical schedulabilty tests have been formulated for systems that employ fixed priority, pre-emptive scheduling. If the execution characteristics of each task are known then the test can be used at design-time to determine whether the task set is schedulable (whether an execution order exists that enables all tasks to run and to meet their deadlines). The execution characteristics that must be known for each task are typically task execution time, task blocking time (the time for which a task is blocked awaiting communication and synchronisation with other tasks) and the deadline. One of the earliest, best known, but most restrictive schedulability tests for RTSs is the rate monotonic test [LIU73]. The rate monotonic approach applies to periodic tasks, and requires that the priority assigned to a task is inversely proportional to its period. Since an underlying assumption of the rate monotonic approach is that a deadline is equal to its period then, as indicated above, priority is related to deadline. A more recent approach to schedulability testing, that relaxes many of the restrictions of the rate monotonic approach, is response time analysis [AUD91].

On-line scheduling is beneficial from a number or perspectives. Development and maintenance is easier since tasks are modelled on their behavioural characteristics rather than intricate timing needs. However problems associated with on-line scheduling

include the issues of task synchronisation, priority inversion and deadlock. These will be discussed in Section 1.6.2. Cyclic executives can be used instead of on-line scheduling and are popular because of their predictability. It is argued that for satisfying timing constraints in HRTSs, predictability of the system's behaviour is the most important concern. As such static scheduling is often the most practical way of providing predictability in a complex system [XU93].

1.5.1 Determination of Execution Time

Irrespective of whether static or on-line scheduling is used in a HRTS, there is a need to be able to establish the execution time of a task. Almost all non-trivial tasks have multiple execution paths, and so can exhibit a range of execution times. Given that a key issue in the development of HRTSs is the meeting of deadlines under all circumstances, then the longest, or worst-case, execution time (WCET) of a task is the value typically used in schedulability calculations.

WCET can be established by measurement, for example by execution profiling. Since this is essentially a black-box approach, it is not wholly satisfactory for HRTSs, since it can never be assured that the critical or longest path through the code has been observed. Hence, although measurement of execution time has a place within the development of HRTSs, static techniques are often employed to analyse the *object code* in order to search for the critical path so that WCET can be calculated [BUR97]. Searching for the critical path may be done using a brute force search of the code, whilst WCET calculations are often performed using a lookup table to determine the execution time of the compiler-generated machine instructions in the critical path. At such a low level, these techniques are naturally platform specific.

In general, the problem of bounding the WCET of an arbitrary sequential program is unsolvable and is equivalent to the halting problem for Turing machines [KOP97]. This means that an algorithm used to compute the WCET may not be able to produce a result for many general-purpose programs. Therefore HRT programs must be carefully designed and programmed in order that the WCET can be calculated. Programming idioms such as [PUS02B], have been suggested which place bounds on looping

constructs and remove data input dependencies in order to support the calculation of WCETs. In cases where this cannot be done, temporal annotations are added to the code in order that intractable parts of the program can be automatically processed. However, this can introduce significant risk since it is difficult to ascertain that the assumed WCET is a guaranteed upper bound of the actual WCET [KOP97]. For example the programmer may make an incorrect assumption about the system or may fail to consider a peak load or exceptional execution scenario.

In many cases WCETs are pessimistic. This has a number of unfortunate consequences. The capability of the hardware is under-utilised. This means that more expensive hardware options need to be used to reduce the likelihood of future upgrades being problematic. Pessimistic WCET results used in schedulability testing may suggest that deadlines could be missed, however this may be unlikely in reality. However this possibility requires expensive changes to the design or hardware in order that the timing requirements can be proved safe. Finally, operating system overheads are not typically considered by such approaches.

Many of the above problems are ultimately caused by low-level temporal non-determinism. In order to reduce the WCET pessimism and promote the design of systems where safe, automatic, tight and realistic WCET analysis is achievable, all the sources of non-determinism must be understood. This is discussed in the next section.

1.6 Sources of Non-determinism in Real Time Systems

A hardware or software component's behaviour is deterministic if, when invoked under the same conditions, it executes in an identical fashion with an identical outcome. It is temporally deterministic when the above condition is true and the time taken, given the time resolution, is always constant. Throughout this thesis, *determinism* refers to temporal determinism and not behavioural determinism.

Most HRT components will exhibit some degree of non-determinism (for reasons discussed in Section 1.6.1 and onwards), however it is important that this is bounded within reasonable limits. Bounded non-determinism means that the best-case execution

time (BCET) and worst-case execution time (WCET) are known and can be extracted *a priori*. The lower the variance between the WCET and BCET then the lower the non-determinism, the higher the temporal predictability and higher the hardware resource utilisation. To this effect, the variance between the WCET and BCET can be used to define the degree of non-determinism. Figure 1 shows three cases of WCET for some operation.

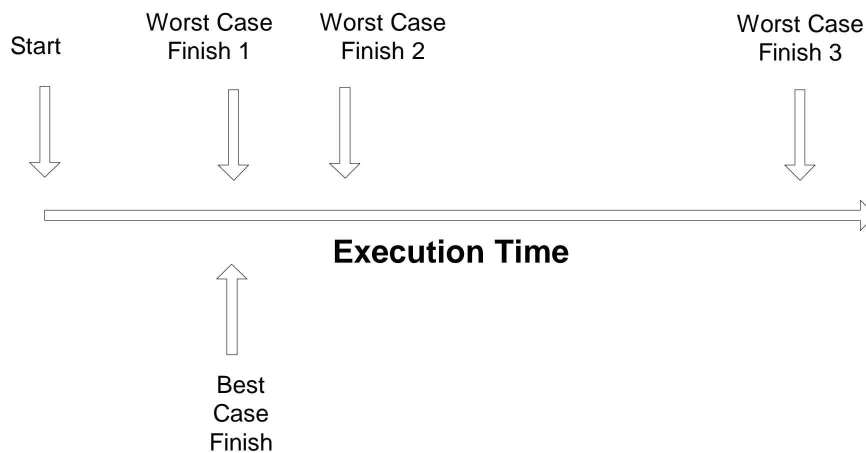

Figure 1 Examples of Worst Case Execution times.

Worst Case Finish 1 is the same as the Best Case Finish, and is therefore highly deterministic which results in very high hardware utilisation. Worst Case Finish 2 has a relatively short distance from the Best Case Finish time, and therefore has relatively low levels of non-determinism. Worst Case Finish 3 has a much larger temporal variance from the Best Case Finish time, and as such has high levels of non-determinism that results in low hardware utilisation.

There are various sources of non-determinism within computer systems and this applies to all levels including programming language, programming constructs, runtime, operating system and hardware. The following illustrates a number of these.

1.6.1 *Programming Language and Usage*

There are several sources of non-determinism commonly encountered in many programs. For example, loop conditions (such as those shown in Figure 2), may be dependent on input data, perhaps from an external sensor. Such conditions make it impossible to predict how many times the loop is executed and hence the WCET cannot be calculated.

```
1       While ( expr1 && expr2 )
2       {
3          // do something
4       }
```

Figure 2 Input Data Dependency.

Similar considerations apply to recursion, where the terminating condition is usually dependent on input data, and so the depth of recursion cannot be known at compile time.

Run-time allocation and de-allocation of resources, including memory, is generally non-deterministic for two main reasons. Firstly it may be very difficult to guarantee that the resource is available when requested, since another part of the system may be using it. Allocation of resources is state dependent in the sense that it is determined by the sequence and nature of previous allocations and de-allocations [SCH92]. Secondly, the time taken to allocate the resource may vary with time. For example the time to allocate a block of heap memory may vary depending on the size of the block required and the design of the heap and the algorithm employed to search for a free block.

The use of dynamic data structures in a program can introduce non-determinism for a number of reasons, some of which are related to the issues discussed above with respect to resource allocation. In addition, the time taken to traverse, or manipulate the data structure depends on the data structure design, its current size and its current state. It may be possible for this to be bounded in some way, but this is likely to result in pessimistic timing results.

Many programming languages contain features that are not well defined, or which are implementation-specific. Hatton [HAT95] defines 22 unspecified, 97 undefined, 76 implementation specific and 6 locale-specific features of the C language which may result in program faults if not considered properly.

The way in which the control flow in a program is organised can also lead to practical difficulties with the calculation of WCET. It is known in general that the number of paths through a program grows exponentially with the number of consecutive branches in the control flow of the analysed code [PUS02B], and this can lead to serious practical problems in establishing WCET.

It is often suggested in the literature that exception handling plays an important role in the provision of a reliable error-handling facility for RTS [BUR97]. However, there are fundamental sources of non-determinism in many common exception mechanisms. In particular, if an exception is raised in a routine that has no handler for that type of exception, then the exception is typically propagated to the routine's caller and re-raised. This process is repeated until a handler is found, or the top level of the program is reached, in which case the program terminates and the execution environment (for example the OS) must attempt to recover. Given that a routine can be called from many places within a large program, it is difficult to place bounds on the time taken for an exception handler to be located. Hence common exception mechanisms are non-deterministic[3].

There is debate about whether object oriented (OO) techniques are appropriate for HRTSs [NAS04]. There are a number of OO features that can make static analysis difficult and introduce non-determinism into a system. It is also questionable whether OO techniques are actually required for HRTSs. This is further discussed in Section 2.4.

---

[3] It can also be argued that exception handling is an unstructured approach to error handling. "The raised exception is, an up-market goto statement in many ways, and its widespread use can lead to the kind of un-maintainable spaghetti code that high-level languages are designed to avoid" [BUR98].

1.6.2 Operating System

Non-determinism is present in various features at the OS layer. The use of on-line pre-emptive scheduling as described in Section 1.5 can introduce a number of problems. If a resource is shared between tasks, this generally means that only one of them can safely use the resource at a given time [BUR97]. The resource needs to be locked by a task in order to prevent race conditions. If another task then attempts to lock the resource it will *block* until the resource becomes available. Blocking time in general is difficult to bound and many synchronisation primitives do not define the order of task resumption once a resource is unlocked and becomes available again.

Task synchronisation can introduce a problem known as priority inversion that results in higher priority tasks being blocked by lower priority tasks [BRI99]. This increases the WCET of higher priority task and invalidates the assumption that a task can be delayed only by higher-priority tasks. Unbounded priority inversion can occur if a medium priority task, which is not dependent on the shared resource, is scheduled before the low priority task releases the lock[4]. Other scheduling faults such as deadlock can occur, resulting in one or more tasks halting [DEI90].

Virtual memory is a feature offered by many operating systems. It provides the ability to address a memory space much larger than that available in main memory [DEI90]. This allows tasks to execute even though only a small proportion of their code and data is loaded in main memory and means that the system can execute many more tasks than main memory can support [BAC86]. A task uses virtual addresses instead of actual physical addresses. Typically the OS divides the virtual memory space into a set of pages that contain the virtual memory addresses. A number of pages are held in main memory, however, some may be stored in secondary storage (for example a hard drive) until it is needed. When a task executes and references a virtual address, it is translated into a physical address using a set of internal mapping data structures. If the associated

[4] It can be argued that multi-tasking bugs such as priority inversion and deadlock can often be the result of complex or poor design including conflicts between task decomposition, task priorities and resource usage.

page for the virtual address is not in main memory, the OS copies the page into main memory at a free location specified by the mapping data structures. If space needs to be made for the page in main memory, another page is transferred from main memory to the secondary storage device. The time taken for a memory access depends on whether the page containing the virtual memory address is currently held in main memory or secondary storage. The memory access is state dependent and is based on the sequence of previous memory accesses.

1.6.3 Hardware

Various processor features are non-deterministic. This is generally because modern processor design aims to optimise the mean throughput, typically by making the common case fast [HEN96]. Hardware features that are non-deterministic are discussed below.

Instruction and data caches rely on temporal and spatial locality of programs in order to improve average case memory access times. There are various cache architectures that can be employed and these are discussed in [HEN96]. Cache memories work on the principle that a small section of a program is likely to be accessed more than once within a short period of time. A *cache miss* occurs when an instruction or data item is referenced by a program but is not in the cache. The instruction or data item needs to be *loaded* into the cache. When a cached instruction or data item is modified, it typically needs to be written back to the cache and also to main memory.

The use of locality assumption algorithms in cache designs leads to non-determinism because of the possibility of a cache miss. For example, instruction caches exploit locality because instructions are generally executed sequentially and referenced multiple times within programming loops. However, program execution can vary greatly depending on whether the instruction being fetched is currently in the cache or not. This depends on the program’s preceding (and potentially very long) sequence of operations. Context switches and interrupt service routines may induce cache interference by replacing the instructions of one task with those of another.

A processor can increase instruction throughput via the use of a pipeline. Some vendors may claim that by pipelining the instruction cycle, one instruction per clock tick can be completed. However in reality, pipeline hazards such as those outlined by [HEN96] often prevent this. Intrinsic hazards such as structural, control and data stalls[5] mean that a 'bubble' is inserted into the pipeline, resulting in a stall. Extrinsic hazards such as a branch or context switch cause the pipeline to become completely invalid and a flush and subsequent reload is required. The points and frequency with which these hazards occur is dependent on the flow of program execution and is non-deterministic.

A processor using a pipeline can (in the best-case scenario) complete the execution of an instruction on every clock cycle. However, a superscalar processor can execute more than one instruction on each clock cycle. This is possible because they employ multiple functional units so that instructions can be processed in parallel. For example, a superscalar processor may employ multiple arithmetic logical units (ALUs) so that a number of ALU operations can be executed in parallel. Typically, additional hardware called a *dispatcher* is used to decide which instructions can be executed in parallel, and to manage the allocation of the functional units. However, the use of superscalar processing introduces non-determinism since the dispatching of instructions depends on the program's preceding sequence of operations and the decisions made by the dispatcher.

A Direct Memory Access (DMA) controller is employed to transfer data between two devices or memory banks with reduced CPU involvement. The CPU is able to execute non-I/O instructions at the same time as the DMA is in operation. However its use can result in non-determinism, since the DMA controller may lock the data bus when the

---

[5] A structural stall is used when the pipelining of certain instructions causes a hardware resource sharing conflict. A control stall is used when a branch instruction is executed; otherwise subsequent instructions that overlap with the branch instruction may complete execution even though the programmer intended for them to be skipped. A data stall is used when an instruction uses the result of an earlier instruction; because the execution of the instructions overlap with one another, the second instruction may start to use the result of the first instruction before the result is available.

CPU requires it [MAN01]. This is likely to result in a delayed execution of instructions, depending on the frequency and size of the data blocks being transferred.

Even at the level of the Arithmetic Logic Unit (ALU) there is non-determinism. This is due to algorithms that are data dependent, such as those often used in multiplication, division and shifting. In the case of multiplication, Booth's algorithm is commonly employed, due to the fact that it handles twos complement multiplication with ease. More importantly, it can significantly reduce the number of partial products (by recoding the multiplier values), which in turn reduces the number of additions required for a given multiplication. Booth's algorithm for multiplication improves the average case by reducing the number of steps required [KOR01]. However the multiplication may take a variable number of clock cycles depending on the operand values. The shifter is another example where the execution time is dependent on the operand data. A shift left by two places typically consumes fewer clock cycles that a shift left by eight places. Again, the number of steps required is dependent on the operand data and therefore non-deterministic.

Processors may employ internal buffering of data, for example multi-window registers [KAT84]. In certain scenarios, depending on program flow, these buffers may need spilling into main memory, which introduces non-determinism. Stack-based computers may use on-chip stack sequential last in first out (LIFO) buffers. If the stack becomes full during program execution, then some of its contents need to be transferred to main memory. A number of stack spilling algorithms can be employed for this purpose. However the point at which spilling is required depends on the flow of program execution and is non-deterministic.

1.7 Implications for HRTSs

Non-determinism impacts the design and programming of HRTSs in a number of ways and includes various aspects of the system. The following section discusses the implications that this presents from both a software and hardware viewpoint.

A number of steps can be taken to control software non-determinism. A language subset can be created in order to remove any undefined and implementation specific features. Aspects of the language that are considered non-deterministic can be omitted. For example, certain OO features and exception handling features may be removed from the HRT subset. An example of a language subset for HRTSs is Spark Ada [BAR03].

Programming idioms can be employed which remove input data dependencies from programming constructs, and these are discussed in [PUS89]. This essentially means that looping and recursion constructs are bounded instead of being dependent on program or device state. Resource allocation can be simplified so that all resources are allocated once only when the system is started. Program flow can be simplified in order to aid WCET analysis. Single path programming techniques outlined by [PUS02B] can be used in order to prevent input data dependent branches.

A number of further constraints can be made at the OS layer. Kopetz argues that HRTSs are naturally time triggered as opposed to event driven [KOP97]. If the HRTS is simple enough, it would seem sensible to use a cyclic executive instead of an on-line scheduler. However, Burns and Wellings acknowledge the problems inherent with time triggered approaches and therefore advocate the use of on-line scheduling [BUR97]. Priority inversion discussed in Section 1.6.2 can be bounded by using a priority inversion avoidance algorithm, such as the priority ceiling protocol or the priority inheritance protocol [SHA90]. The use of RMA discussed in Section 1.5, dictates that the number of tasks must be fixed. This together with the fact that memory is usually allocated once during system start-up means that dynamic memory management is avoided. The OS itself must be developed in the same manner as a HRT application and so the choice of OS is generally restricted to those classified as Real Time Operating Systems (RTOSs). The Worst Case Administrative Overhead (WCAO) of every OS service must be known *a priori* so that the temporal properties of the complete system can be determined [KOP97].

There are a number of ways in which non-determinism can be controlled at the hardware level. Caches can be omitted or disabled. At the simplest level, high-speed

memory can be used instead of caches. This can be loaded once at start-up with code that is frequently executed and data that is frequently used. Careful consideration needs to be given to the use of instruction pipelining. Program branching and pre-emption due to context switching or interrupt service routines means that pipelining architectures are usually avoided. Scalar processors that execute instructions sequentially are used instead of superscalar processors. Hardware features that can cause bus contention (for example DMA) and features such as automatic buffer spilling are avoided.

ALU algorithms that improve the average case, for example Booths' multiplication algorithm, are of no use in HRTSs. In this example, operand values which require the most clock cycles to execute are assumed every time a multiply instruction is performed. Alternatively simpler processor designs or those that employ operand independent optimisations may be used.

1.8 Summary

RTS can be classified as either hard or soft. HRTSs are very different from SRTSs in that deadlines must be met under all runtime scenarios. Many HRTSs are also safety critical. An important aspect of ensuring that deadlines will be met is schedulability testing. If a system is simple enough, a cyclic executive can be used to model concurrent system tasks and therefore schedulability testing is done by virtue of the fact that an off-line schedule can be built. Schedulability testing for an online scheduler is more complex and a commonly used technique is RMA. Other online scheduling problems such as unbounded priority inversion and deadlock must be considered.

A key parameter for schedulability testing is the execution time of each task. The BCET or average execution time is of no use since deadlines must be met under all possible runtime scenarios including the worst case. Hence the WCET of the task is used. There may be a large variation between the BCET and WCET, usually due to temporal non-determinism in various aspects of the system including language, program, OS, runtime and hardware.

Non-determinism can be controlled at the language, program and OS layers via a disciplined approach to development. Simple hardware designs that execute instructions in sequence are used instead of complex designs aimed at improving average instruction throughput. Caches, DMA and virtual memory are usually omitted or disabled. Whilst this may significantly reduce the average hardware performance, techniques that speed up the average case are of no use in HRTSs.

2 Using Java for Hard Real-Time Systems

2.1 Introduction

This chapter seeks to evaluate the suitability of Java and its derivatives for use in the programming of HRTSs. The chapter begins by defining a set of requirements that must be met by a language that is to be used for HRT programming, and discusses the extent to which contemporary languages meet these requirements. The suitability of Java and its real-time variants is then investigated. This then leads to a statement of research objectives for this thesis.

2.2 HRT Support Provided by Contemporary Languages

The nature of RT and HRTSs was discussed in Section 1. From this, and also from [BUR97] a list of language features or characteristics that are important in the context of HRTSs can be identified. The characteristics and features are determinism, reliability, modularisation, concurrency, device interfacing and error handling. These will now be discussed briefly.

Many HRTSs have very high reliability requirements, and so must exhibit a very low failure rate. It has been found empirically that some languages lead to greater numbers of errors than others [GER03]. For example [GER03] found a difference of two orders of magnitude in the rate of occurrence of errors in C programs when compared with SPARK Ada programs. The conclusions formed were based on “metrics found from various programs” [GER03]. It can be argued that factors other than the language have a greater impact on error rates. However the author acknowledges the general theme that some languages make the development of HRTSs easier and less likely to contain errors. Hence the features offered by a language, and those that are absent, can have a significant effect on the reliability of programs written using the language.

As discussed in Section 1.5, determinism is important as it allows the system to behave in a consistent and predictable manner. A HRTS must be temporally deterministic in all situations, and so language constructs must not only behave in a deterministic manner but must also enable programs to be written that execute with constant and repeatable

timing characteristics. Although elements outside the programming language can introduce non-determinism at runtime (for example the operating system), there are certain language features that are intrinsically non-deterministic, or prone to usage in a manner that leads to non-determinism (see Section 1.6.1). For these reasons, it is common for language subsets to be used for HRTSs.

Strong typing is a language feature that is often associated with reliability. The term relates to the rigour of the language's type compatibility rules, whether the rules are explicit or implicit, and whether they are applied statically or dynamically. Strong typing enables a compiler to identify some errors that can only be found through inspection, testing or debugging in less strongly typed languages. Although more strongly-typed than some languages, C is regarded as weakly-typed since it supports a number of features, for example, implicit type conversions, which enable type-unsafe operations to be performed. Ada is perhaps the most strongly typed language in commercial use for the development of RT software. For example Ada does not provide implicit conversions between numeric types. It could be argued that the difference in the strength of typing between C and Ada is one factor contributing to the difference in the error rates observed by [GER03]. With particular respect to object oriented languages, it can be argued that there is a type safety loophole in any language that supports polymorphism[6].

Pointers provide programmers with a great deal of flexibility, especially with respect to the control of peripheral devices, which is a key task in RTS. However, some commentators [BUR97] argue that pointers, in their most general form (for example as found in C/C++) are unsafe as their misuse can result in intermittent bugs that cannot be detected by the compiler and also prove difficult to detect for the programmer. Other languages, notably Ada and Java, provide a more restricted form of pointer in an attempt to compromise between flexibility and safety.

[6] Using a base class reference to access an object derived from the base class means that the accessor is inherently weakly typed.

Device interfacing is a low-level and error-prone activity that typically requires access to device registers and the coding of interrupt service routines (ISRs). Ideally, access to registers should be performed at a high-level and be type-safe. Although pointers in C provide a flexible access mechanism, they are not type-safe. C does not support the coding of ISRs, although C compilers often allow programmers to write ISRs in C and link them to an interrupt vector. In contrast, Ada provides high-level, type-safe approaches to register access, via representation clauses, and to the integration of ISRs via protected procedures.

Modularity has been found to be an important factor in improving the reliability of software. It is also important for maintainability. Modules encapsulate data and the operations that manipulate it. Interaction with other modules is via well-defined interfaces. Using modular programming promotes strong cohesion and low coupling in programs. Some languages, for example Ada and Java, provide direct support for modular programming in terms of programming constructs. Other languages, such as C, do not provide as much support, although modular programming can be effectively achieved through a disciplined use of the language.

In Section 1.5 it was stated that cyclic executives are commonly used for HRTSs. If the system is simple enough, then this is likely to be the best approach. Therefore support for concurrency is not necessarily a primary language requirement for programming HRTSs. However, on-line scheduling is receiving attention in the research community in order that the benefits of an abstract model of concurrency can be employed by more complex systems. The notion of a process within a real-time programming language increases the expressive power of the language [BUR97], and so makes concurrency easy to express. If processes are included within a language, then facilities to support inter-process communication and synchronisation must also be added. However, concurrency and blocking communications/synchronisation mechanisms can introduce many problems, such as unbounded priority inversion, deadlock and livelock, and caution must be exercised when using such features. Some languages support concurrency directly, for example Ada and Java. Although others like C and C++ do not, concurrency can still be used via the facilities of the underlying OS.

2.3 Benefits of using Java for General Software Development

The previous section illustrated how modern programming languages support the needs of HRTSs. Java provides support for many of these needs. It has been acknowledged that the use of Java often results in stable, high-quality programs with a reduced number of faults. Java provides interfaces rather than multiple inheritance. Interfaces provide similar abilities as abstract base classes or *mixins* described in [BOO94]. An interface is essentially a specification that describes the method signatures that an implementing class must provide[7]. This promotes design by composition instead of inheritance [COA96]. Interfaces also avoid problems associated with multiple inheritance such as repeated inheritance as described in [BOO94].

Java is generally regarded as a strongly typed language that helps in identifying programming bugs during compilation. Concurrency is supported directly by the language. As discussed in Section 2.2, these are beneficial language constructs for developing HRTSs. References are provided instead of pointers and it is argued that this reduces the errors usually found in C and C++ programs. The JVM (which is discussed more in section 4.3.1), uses automatic memory management in the form of a garbage collector instead of requiring the programmer to explicitly de-allocate memory. This means that memory management is easier and results in fewer errors such as memory leaks and dangling pointers. Unlike C++, which is an evolved and overly complex language, Java is well defined and simple [SUNA]. As a result many developers claim that Java is much easier to learn than C or C++ and that they can be much more productive using it. A program written in Java is platform independent. By porting the JVM for each supported platform, in theory, there is no need for the application to consider the underlying processor architecture or OS. A large set of standard Java class libraries are available and the interfaces to these libraries remain the same regardless of the implementation, which allows re-use of learning and increases productivity. Whilst Java offers a number of substantial benefits for general software development, there are

[7] Java Interfaces can also include constant values which can be read by an implementing class.

a number of problems when specifically considering HRTSs. A number of the key issues are now discussed.

2.4 Problems with using Java for Hard Real-Time

As stated in Section 1.1, many HRTSs are safety critical. As such regulatory standards (for example DO178B[8]) should be considered. A NASA report suggests that some OO languages have features that could make it difficult or impossible to satisfy the requirements of DO178B [NAS04]. For example, some of the concerns include the use of polymorphism.

In common with other OO languages, subtype polymorphism in Java is achieved via the use of inheritance and method tables. When a polymorphic function is called, it may be done using a base class reference and the type of object may be unknown. The object can be an instance of any class in a hierarchy. In order that the correct function is called, a level of indirection is used so that function binding is performed at runtime. Hence this is called dynamic binding. Each class is provided with a method table, sometimes called a *vtable*[9].

Dynamic binding provides a number of flexible programming facilities:

1. The called method does not need to be implemented by the class of object invoking it. Instead, a super class in the hierarchy may implement it. This is inheritance.
2. A sub-class can override a base class method in order to provide an alternative implementation. This is overriding.
3. A new class can be added to a hierarchy and existing code may not necessarily need to be re-compiled.

---

[8] DO178B is a certification standard for commercial aircraft development in Europe and the United States. A missed deadline for some systems (referred to as types A and B) could lead to catastrophic consequences.

[9] The term vtable stands for *virtual method table* and is commonly used to describe method tables in C++ implementations.

However a number of problems are manifested with regard to HRTSs. Dynamic method dispatching means that execution flow depends on the type of object used for the call. However this information may only be known at runtime. This means that when searching for the longest path during WCET analysis, an increased number of paths need to be considered depending on the possible set of classes in a hierarchy. In order to restrict the search paths to those that are possible during execution, programmer annotations may be required which increases the risk of analysis errors.

As discussed above, dynamic binding allows method overriding. Class method overriding is often used to change the behaviour of an inherited method and may be regarded as poor design [COA96]. Method overriding can also lead to certification problems in HRTSs as the base class contains deployed, but unused code. This is termed *dead code*, and some safety standards such as DO178B prohibit dead code.

It appears that polymorphism is not required for most applications. Booch cites studies by Deutsch suggesting that polymorphism is not required in 85% of method calls [BOO94]. Assuming that this study relates to desktop software where polymorphism is more likely to be used, it could be argued that polymorphism is not required for HRTSs.

Java relies on automatic initialisation of static class variables[10], which is done via a special *clinit* method. It is called by the JVM the first time that the class is used. This may be the first access to a static field, a static method or the first time that an instance of the class is created. This means that the execution time for an operation is longer the first time if it is the first use of the class. As well as introducing non-determinism, automatic initialisation increases the possibility of faults due to circular static initialisation dependencies. Requiring that static fields be explicitly initialised during an initialisation phase can circumvent such problems.

Garbage collection is an important Java runtime feature. However it can be argued that garbage collection contributes significant levels of non-determinism and also greatly

---

[10] Static class variables may otherwise be referred to as static fields. They belong to a class rather than an object instance.

increases the complexity of the runtime system. In Section 1.7, it was stated that HRTSs simplify memory management by allocating memory once during a start-up phase, therefore removing non-determinism and making memory usage simple to calculate. Therefore garbage collection is unnecessary for HRTSs.

Section 1.2 described HRTSs as being inherently platform specific. This is because devices are required to be controlled and monitored at a low level and WCET is naturally tied to a specific platform. The Java language is incapable of low-level device programming. The Java Native Interface must be used to call and interface with native C code [SUND]. The C code is then used to control and monitor the external devices. This approach increases complexity since there needs to be a mapping between Java types and method invocation and the corresponding equivalents for C and the native platform.

Java employs dynamic class loading which allows the class files for an application to be loaded and linked as and when required. However dynamic class loading in HRTSs is inappropriate and unnecessary. It is inappropriate since many HRTSs are safety critical, and hence the loading of classes from outside the system would invalidate the integrity of the system. It is unnecessary, since HRTSs are closed due to the fact that they are employed for very specific and defined purposes and are not general-purpose computing platforms. It would be impossible to calculate the WCET for a class during loading and execution, as the class is unknown. It is also unlikely that the WCET for the dynamic class loading and bytecode verification could be determined due to its high complexity and class file data dependencies.

The Java specification prevents the use of schedulability testing such as RMA. This is because low priority threads may pre-empt high priority threads to avoid starvation [GOS00]. This makes it impossible to verify that Java threads will meet their deadlines. Additionally Java does not mandate the use of priority inversion prevention protocols [GOS00] and therefore unbounded priority inversion or deadlocks may arise. Monitors introduce non-determinism due to the use of wait set queuing. A JVM implementation is free to decide randomly which thread to schedule in a monitor wait set [GOS00]

instead of making a decision based on its priority. There are a number of subtle problems with the Java memory model [JSR133].

It is clear that in its standard form such as [SUNB], Java is not suited for HRTSs development. Hence, specialised variants of Java have been developed, for example, the Real Time Core Extensions (RTCE) [JCON] and the Real Time Specification for Java (RTSJ) [JSR01]. The RTCE is implemented as a separate class library which runs alongside non-RT code on a conventional non-deterministic JVM. As such the RTCE does not address the problems of non-determinism at the JVM level. The approach of the RTSJ is different. The RTSJ extends various aspects of the Java Language Specification [GOS00] and the Java Virtual Machine Specification [LIN99]. The RTSJ is discussed in the next section.

2.5 Real-Time Specification for Java (RTSJ)

The Real-Time Java Expert Group (RTJEG) defines the Real-Time Specification for Java (RTSJ) and its purpose is to enable the use of Java in RTSs. The RTSJ remains compatible with standard Java. A RTSJ JVM must not prevent the execution of properly written non-real-time software designed for any other JVM implementation [DIB02]. As such, a number of features are added to standard Java, which cover areas such as scheduling/threading, memory management and device programming.

A fixed-priority pre-emptive scheduler is supported with a minimum of 28 distinct priority levels above the 10 provided by standard Java. Programmer-defined scheduling algorithms can be used. This allows scheduling approaches such as round robin with interrupts to be used [SIM99]. Priority inheritance is supported in order that priority inversion can be bounded and deadlock can be eliminated. This means that schedulability testing such as RMA can be carried out. A RT thread has a number of parameters such as priority, execution time, deadline and type. The type can be either periodic or sporadic. RTSJ is backwards compatible with standard Java in that regular non-real-time threads can be executed as well as RT threads.

Garbage collection is supported, though alternative approaches to memory management are also provided. Objects can be constructed in *scoped memory* where objects exist for a fixed duration of program execution. Objects can also be created in immortal memory which means that they exist for the entire duration of the program's execution. Objects can be placed at specific physical memory locations and raw memory access can be achieved using peek and poke operations. These features allow memory-mapped devices to be programmed. As discussed in Section 1.2, this is an important requirement for HRTSs. Section 1.6.1 highlighted that object creation may be non-deterministic for a number of reasons, one being due to the time taken to search the heap for a block of free memory. RTSJ allows memory to be allocated during object creation in constant time by employing heap management techniques similar to those discussed in [SIE02]. Whilst a number of memory types are provided this unfortunately leads to increased complexity as the JVM needs to ensure that assignment operations for objects in different memory areas are legal and valid [DIB02].

The RTSJ provides asynchronous events, which allow external interrupts to be handled. Asynchronous events provide similar behaviour to interrupt service routines (ISRs) in many RTOSs. Asynchronous Transfer of Control is supported, which allows threads to communicate asynchronously with one another. This is implemented as a signal sent from one thread to another, causing the receiver to branch unconditionally. A high-resolution time library for accurate timing is also supported.

It is the author's opinion that whilst the RTSJ may be appropriate for some SRTSs, it is unsuitable for HRTSs. Additionally, as discussed in Section 1.4, the JVM is a large and very complex program. The RTSJ does not identify and address non-determinism in the JVM or Java language. As discussed in Section 1.1, the consequences of faults in HRTSs are usually much more severe than faults in desktop and SRTSs and may include loss of life. As such HRTSs usually employ language and runtime subsets in order to remove unnecessary and undesirable features and reduce the risk of faults. Features that are non-deterministic may also removed in order to simplify WCET analysis. Section 1.2 stated that many HRTSs are safety critical and often driven by regulatory safety standards. In order to ensure that the system will behave as intended

these systems often require static analysis described by [GER03] as well as very high-test coverage. As described in Section 1.3, developing HRTSs is difficult, time consuming and very expensive which means systems are made as small and simple as possible. In order to make Java suitable for HRTSs a HRT Java profile has been defined by [PUS2A].

2.6 Puschner, Bernat and Wellings HRT Profile

As discussed above, the RTSJ is inappropriate for HRTSs. Puschner, Bernat and Wellings have defined a profile specifically for developing HRT Java programs [PUS02A] [11]. The motivation is to provide a smaller and simpler runtime with improved temporal predictability by restricting RTSJ features.

Programs written using the HRT profile are organised so that execution is split into two separate phases, the initialisation and mission phases. The initialisation phase is used to set the system up and may include application code that is un-analysable or temporally unpredictable. This includes dynamic class loading and bytecode verification. The execution phase is entered upon completion of the initialisation phase and is where the time critical parts of the HRT application are executed.

The garbage collector is omitted and all resources including objects and threads must be created during the initialisation phase. Such activities cannot be done once the mission phase has been entered. The garbage collector is removed from the profile in order to remove the temporal unpredictability associated with it as well as dynamic object allocation and de-allocation.

The threading system is restricted in a number of ways. A priority-based pre-emptive scheduler with ceiling priority is used and the number of threads in an application is fixed. Priority ceiling is supported by the *synchronized* Java keyword in order to prevent unbounded priority inversion and deadlock. Threads are also assigned temporal attributes such as WCET, period and deadline. This is so that an exception can be

[11] Other more recent work comparable with the Puschner profile includes Ravenscar-Java [KWO02] and HIDOORS [HUN04].

thrown if a deadline is missed. In order to simplify thread communication and remove queuing threads may only interact via shared data [PUS02A]. *Wait*, *notify* and *notifyAll* methods from the *java/lang/Thread* class are removed.

The Puschner Profile improves the RTSJ by making it more suitable for HRTSs. However there are still a number of features that remain unpredictable or contribute significant levels of non-determinism. The Puschner Profile has been somewhat sparing when considering the requirements of HRTSs and a number of further constraints are required in order that it can be used for HRT development. These are discussed in the next section.

2.7 Critique of the Puschner Profile

The Puschner Profile includes dynamic class loading. Due to the unpredictability of class loading, it is done during the initialisation phase. However, the profile requires that all classes are known before the system starts and all classes are completely loaded during the initialisation phase [PUS02A]. Because the profile requires that all classes are known a priori, this means that there is no real need for dynamic class loading. An alternative approach such as *romizing* can be used, and is discussed further in Section 4.5. As discussed in Section 2.4, dynamic class loading in HRTSs is inappropriate and unnecessary and should be removed.

The initialisation phase is used to contain non-analysable and non-deterministic Java features, for example dynamic class loading and bytecode verification. However, the mission phase is dependent on the initialisation phase, because it executes directly after the initialisation phase. The mission phase relies on the initialisation phase to put the system into a known safe state. Error detection before the mission phase is absolutely crucial [PUS02A]. Therefore, it is suggested that the initialisation phase should be treated as part of the mission. It is the author's view that it is unacceptable for the initialisation phase to be unpredictable, un-analysable or less carefully scrutinised than the mission phase. If the initialisation phase is developed using less rigorous techniques than the mission phase, then there is reduced confidence in knowing when the mission

phase will begin or whether it will be started in the correct state[12]. HRTSs may have initialisation phases interleaved with mission phases in order that the system can change modes. In these cases it is essential that the initialisation phase be treated as part of the mission. It also appears that stack depth and memory analysis could not be statically analysed for the initialisation phase.

As discussed in Section 2.4, the automatic initialisation of static class variables and the execution of static code blocks is done by side effect by the JVM automatically calling the *clinit* method. This is done the first time a class is used by a program. This means that execution time for the affected operation is variable. Complex or circular dependencies may exist between *clinit* methods that can make programs error prone as well as making WCET difficult. It is usual for HRT applications to ensure that all static class variables are explicitly initialised at start-up rather than relying on initialisation by side effect. However the Puschner profile does not address the specific problems associated with static initialisation.

As discussed in Section 2.4, the Java class libraries deployed alongside the JVM are too big for most HRTSs. They are likely to be non-deterministic since they were not intended for use in HRTSs. Some safety critical standards prohibit the inclusion of unused library code in safety critical systems and must therefore be omitted.

As discussed in Section 2.4 various OO features for example, polymorphism may introduce non-determinism. However the Puschner Profile does not address these potential problems. The execution times of all operations of the JVM have to be bounded [PUS02A].

2.8 Research Objectives

The Puschner Profile specifies a Java language variant, along with idioms that are better suited to HRT than the RTSJ. However, as discussed in Section 2.7, the profile still

[12] At the Open Group [OPE01] meeting in May 2005, a participant in the audience explained that the "standard procedure" taught to airline pilots who are experiencing "flaky behaviour" of flight instruments is to cycle the power to the flight instruments [NIL05]. This illustrates one possible scenario that could occur if a non-HRT initialisation phase is used.

contains some non-analysable and non-deterministic features, which make it unsuited to HRTSs development. Furthermore, irrespective of the profile, a deterministic JVM is required in order that HRT applications can be executed. As discussed in Section 1.1, many HRTSs are also safety critical; hence a practical solution must also be suitable for the development of safety critical systems. It appears that current JVM implementations do not consider temporal predictability and are unsuitable for HRTSs. Hence the objective of this work has been to implement a JVM suitable for HRTSs. A full and complete implementation is unrealistic. However it is feasible to implement a JVM that will provide support for programming simple HRT applications.

2.9 Summary

Java provides a number of features required of a modern language for developing HRTSs. There are a number of additional benefits with Java, such as it is well defined, easy to use and popular in industry and academia. However, there are a number of Java features that make it unsuited to developing HRTSs.

In order that Java can be made more suited for developing HRTSs a number of RT variants have been proposed. The Puschner profile improves and simplifies the Real-Time Specification for Java by removing some of the unnecessary features and splitting execution into two separate phases in order to localise remaining non-analysable and non-deterministic features. However it is argued that the Puschner profile is still unsuitable for HRTSs.

Regardless of the specified profile, a deterministic JVM is required in order that HRT applications can be executed. There has been little research into the issues of implementing a JVM suitable for HRTSs. The objective of the research has been to implement a HRT JVM that will provide support for programming simple HRT applications.

3 JVM Implementation Approaches

3.1 Introduction

Before discussing the design and implementation of a JVM for HRTSs, it is necessary to understand the different ways in which a JVM can be implemented while paying particular attention to the needs of HRTSs. This section describes ways in which JVMs can be implemented. An implementation approach is then selected for the development of a JVM for HRTSs. The work required for the implementation is then discussed.

3.2 Approaches

There are a number of ways in which a JVM can be developed [COA01A]. These include software interpretation, just in time compilation (JIT), ahead of time compilation, hardware accelerator and hardware JVM. Each of these approaches has a number of advantages and disadvantages. These are now discussed in relation to the needs of HRTSs.

3.2.1 Interpretation

Many JVMs employ software interpretation, an example being the Kilo Virtual Machine [SUNC]. The software interpreter usually consists of a main interpreter loop, variables which represents the internal registers and an area of memory which represent the stacks (the Java stack is discussed in Section 4.3.1). The interpreter loop fetches the next Java bytecode (JBC) for the current method and depending on its value, the appropriate JVM routine is invoked in order to execute the required behaviour of the JBC.

Although this approach is the simplest in terms of implementation, it is also the slowest. Most benchmarks suggest that a Java interpreter will execute code at about one-tenth the speed of compiled C code [MAN98], [COAT01B]. JBC is very compact when compared to native machine code. This results in a smaller memory image for the Java application than for compiled C code. However the JVM interpreter consumes memory, since it is generally a large and complex C program.

3.2.2 Just In Time Compilation

Just in time (JIT) compilers, translate the Java byte code into native processor code as the program is executed by the software interpreter. The motivation for this is to overcome the performance overheads associated with software interpretation whilst keeping the Java class files small[13].

JIT does improve the execution time of methods, but suffers from a number of serious drawbacks. The first is that compilation time must be added to method execution time. If the compiled code is cached, then this overhead only occurs the first time that the method is executed. If the code is not cached, then the method execution time is the sum of actual execution time plus its compilation time. Given that execution times for compilation are unlikely to be known or shown using WCET analysis, then this approach is clearly unsuitable for HRTSs. Due to its complexity, it is unlikely that other static analysis techniques could be applied with this approach.

3.2.3 Ahead of Time Compilation

An effective way of significantly speeding up the execution of Java programs is to use ahead of time compilation, which is the approach used by most programming languages. The translation of JBC to native machine code is done during development, so the WCET can be determined using traditional techniques. Ahead of time compilation is clearly the most appropriate software execution approach for HRTSs, but portability is reduced.

3.2.4 Hardware Acceleration

A hardware accelerator typically extends an existing processor and can be thought of as a 'hardware JIT' approach. The accelerator translates the JBC into equivalent native code so that the main processor can then execute it. A proportion of the JBCs are directly translated into native instructions. Due to the complexity of some JBCs, particularly the OO instructions, some JBCs may result in the execution of runtime functions. Hardware acceleration is used by Nazomi [NAZ].

---

[13] Java programs are compiled into a platform independent format called a class file. Class files are discussed further in Section 4.3.1.

3.2.5 Hardware JVM

A hardware JVM executes JBCs in the same way that a processor executes instructions, which means that no translation of JBC is required. Several hybrid hardware/software JVMs have been developed, including PicoJava [SUN04], LavaCore [DER01], Moon2 [VUL03] and JEM2 [HAR]. However, none have specifically considered non-determinism or been designed to support HRTSs.

Typically the hardware JVM executes a small set of core bytecodes as native instructions. Complex instructions are usually implemented in micro-code, in software or by using a combination of the two. It is unlikely that the execution unit and entire Java runtime be developed in hardware, and so a hybrid hardware and software approach is the most practical and the one most commonly used. The disadvantage of this approach is the lack of a tool chain, for example a C compiler and debugger. This means that non-Java based programs cannot be executed by the JVM.

From a research perspective, a hardware JVM yields a number of significant advantages. A hardware-based Java execution engine can be developed without considering integration issues with an existing processor and tool chain. Furthermore, natively executing JBCs in hardware should bring performance improvements. When compared with the other JVM implementation approaches, this is the option that minimises non-determinism, since it offers the greatest control. Hence it was decided that the hardware JVM approach would be used for the HRT JVM. The next section provides a high-level discussion of a number of existing hardware JVMs.

3.2.5.1 picoJava Core

The picoJava from SUN is a hardware JVM intended for consumer electronic products that run Java applications [SUN96]. The block diagram for picoJava can be seen in Figure 3.

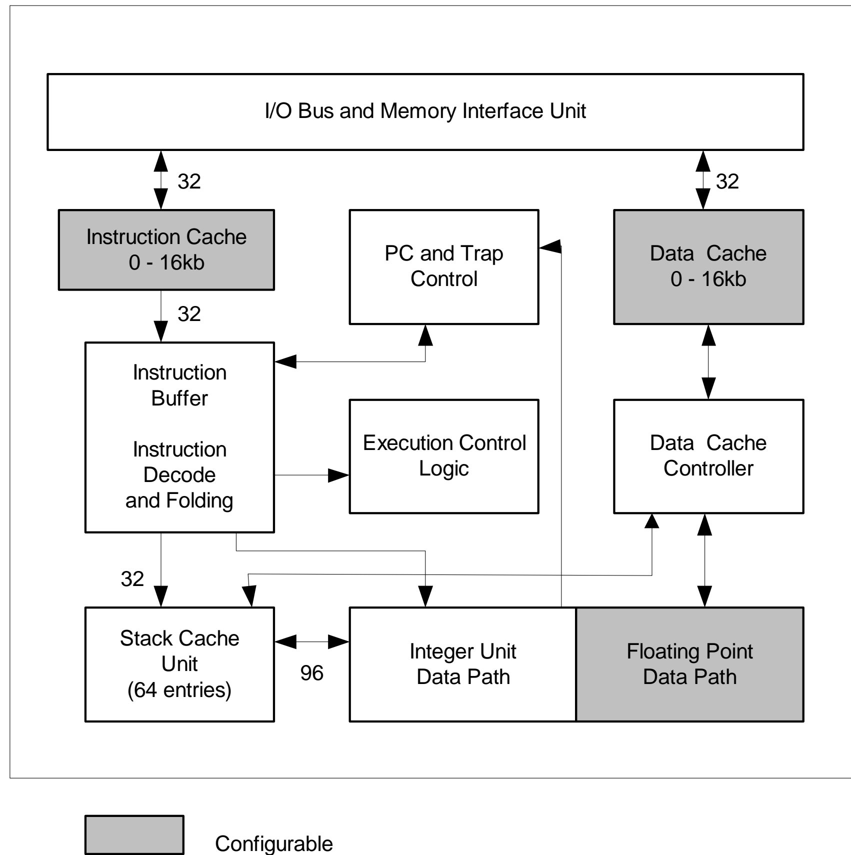

Figure 3 picoJava block diagram (taken from [SUN96]).

Many, but not all of the instructions are implemented directly in hardware. Some are executed via micro-code, while others are implemented in software via the use of software traps. A four-stage instruction pipeline is used to improve instruction throughput.

A Von Neumann architecture [HEN96] is employed, where both instruction code and data are held in the same memory space. The memory space itself is flat; that is, memory addresses may be represented by a single 32-bit integer value. An I/O bus and memory interface unit serves as the link between picoJava and any other hardware that resides on the same chip.

picoJava employs an instruction cache and a data cache in order to improve average case memory access times as described in Section 1.6.3. An instruction buffer is used to

decouple the instruction cache from the main execution unit. The purpose of this is to allow more than one instruction to be processed in a single clock cycle and to help ensure that the pipeline can remain full.

It can be seen that picoJava employs an internal hardware stack (labeled stack cache unit in Figure 3). This is used to store programming constants, operands as well as method arguments, static variables and method activation records. The hardware stack is therefore the equivalent of a register file in a reduced instruction set computer (RISC). The pipeline in picoJava has four stages, comprising of *fetch*, *decode*, *execute* and *write-back*.

picoJava employs support for stack spilling as described in Section 1.6.3. To ensure that a stack overflow does not occur, picoJava anticipates and prepares for stack growth. In the event that a stack cache overflow is predicted, values are spilled into the data cache. Ultimately the stack data may be written to main memory by the data cache controller. When there is room for data in the stack cache, then the values are restored.

picoJava uses an optimization called stack folding. Frequently, an instruction that copies data from a local variable to the top of the stack is followed immediately by an instruction that consumes that data. The instruction decoder detects this situation and effectively folds these two instructions together. This compound instruction performs the operation as if the local variable were already located at the top of the stack.

In summary picoJava is suited towards general-purpose and SRTSs since it contains features that are unsuitable for HRTSs, for example, instruction and data caching, pipelining and stack spilling as discussed in Section 1.6.3.

3.2.5.2 LavaCore

LavaCore [DER01] is a 32-bit Java soft-core aimed at devices such as internet appliances and industrial control systems. Similar to picoJava, LavaCORE executes a large proportion of JBC's using micro-code. However, complex JBCs such as object creation, method invocation, exception handling and synchronization are implemented

using software traps. Floating-point instructions are supported, via an optional module. Figure 4 illustrates the block diagram for LavaCore.

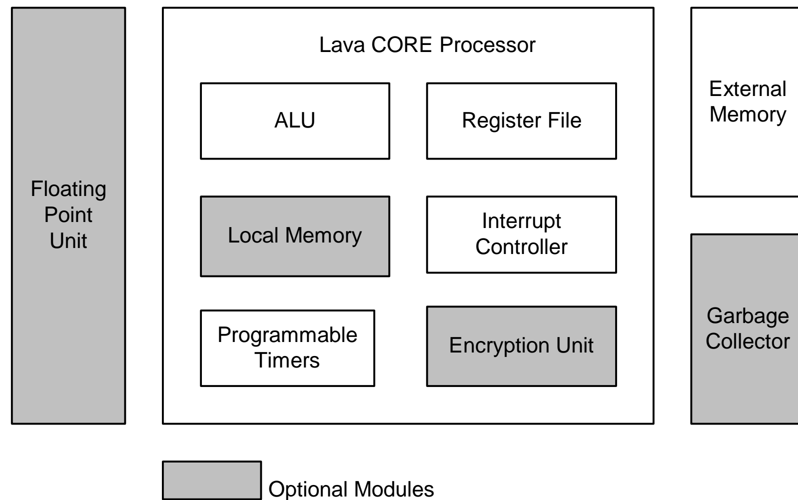

Figure 4 LavaCore block diagram, taken from [DER01].

LavaCore employs a 32 bit dual ported RAM register file instead of internal hardware stack. A pre-fetch instruction queue is used to reduce memory access and the instruction unit comprises of 3, 8-bit registers. One register is used for the JBC and two for the JBC parameters.

As can be seen from Figure 4, optional hardware components can be included such as a floating-point hardware unit, an encryption unit, high-speed internal memory and programmable timers. LavaCore has been designed to be configurable. Using a configuration software tool, the processor can be optimised for a particular application. For example, certain JBC instructions can be omitted or moved from hardware to software if required in order to reduce silicon gate count. LavaCore provides a set of internal instructions that allow memory and external devices to be manipulated.

A romizer is provided as part of a tool-chain. The romizer is used to translate a set of platform independent class files into a JVM specific memory image that can be loaded and executed by the HRT JVM. The romizer is used to initialise the interrupt controller.

See Section 4.5 for a more detailed discussion of romizing. Depending on the size of the application, the memory image may be loaded into high-speed on-chip memory, instead of using instruction and data caches.

3.2.5.3 Moon2

Moon2 is a 32-bit Java soft-core intended for implementation on an FPGA [VUL03]. It is designed for use in mobile phones and set-top-boxes. Figure 5 is a block diagram showing the Moon2 processor.

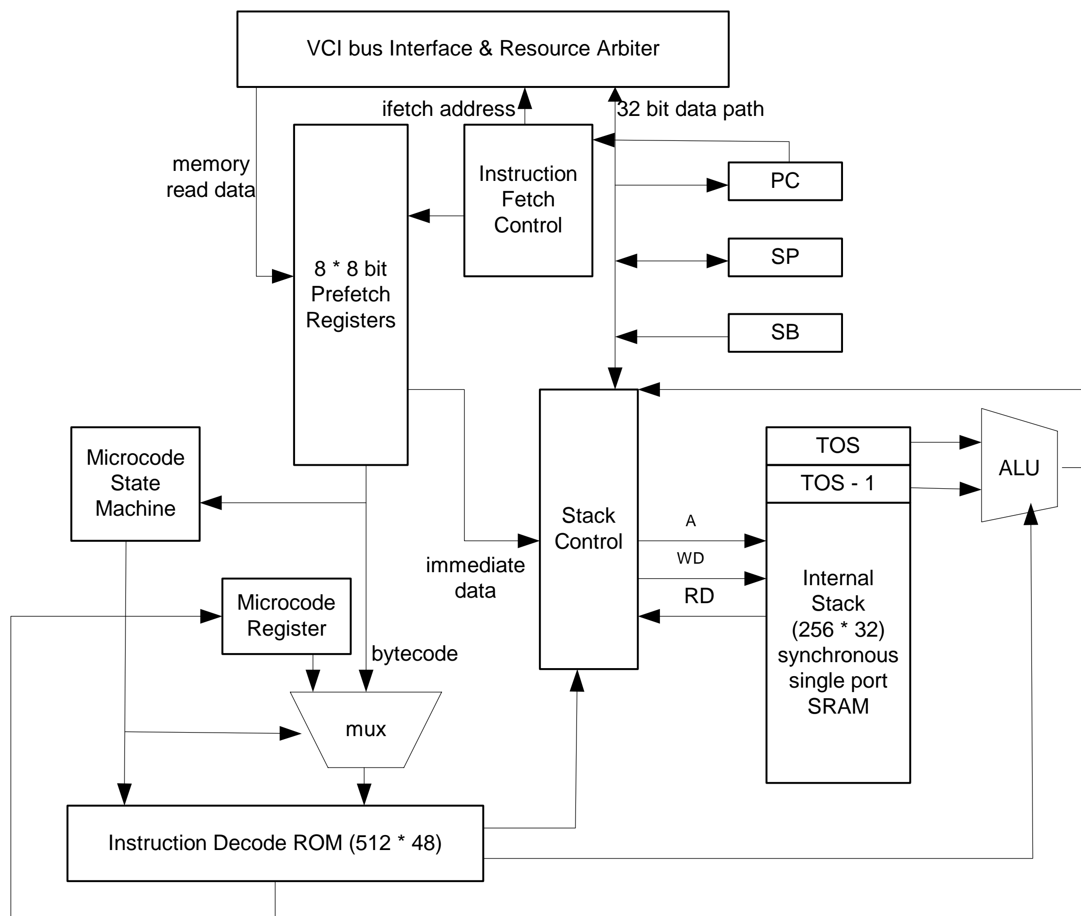

Figure 5 Moon2 Java Processor, taken from [VUL03].

Similar to PicoJava, Moon2 uses an internal hardware stack as opposed to a register file. The most basic JBC instructions are implemented directly in hardware. A large proportion is implemented in micro-code and the most complex JBCs are implemented using software traps. The instruction execution unit fetches bytecodes from the memory location pointed to by the program counter (PC). Typically, more than one complete bytecode is fetched at a time from memory, since the width of the data bus is larger than most bytecode sequences. The bytecodes are stored in a set of pre-fetch registers, which can hold several memory words at a time. The instruction fetch logic attempts to read the next memory word, provided there is room available in the buffer. The current bytecode is decoded and a corresponding micro-code routine held in read only memory (ROM) is executed.

ALU operations operate on the top two elements of the stack. The stack pointer register (SP) points to the top most stack entry in use. The stack base pointer register (SB) is used to indicate the start of a zero based array consisting of local variables and method parameters within the stack. This is used when manipulating local variables and method parameters, for example via the *iload* and *istore* instructions. This is described further in Section 4.3. Similar to PicoJava [SUN04], *stack folding* is performed by the hardware in order to optimise bytecode execution.

A Von Neumann architecture is used, where both instruction code and data are held in the same memory space [HEN96]. The memory space itself is flat; that is, memory addresses may be represented by a single 32-bit integer value. The bus interface unit is responsible for external bus arbitration when accessing memory or other external devices. The tool chain for Moon2 includes a Romizer that produces a memory image of the application and runtime system. This is loaded at start-up by the processor. Dynamic class loading is also supported so that classes can be loaded at runtime.

3.2.5.4 JEM2

The JEM2 processor from aJile [HAR] is intended for low-power, real-time and networked embedded applications, for example mobile phones, telecommunications, automotive and industrial automation. JEM2 is an enhanced version of JEM1, created in

1997 by the Rockwell-Collins Advanced Architecture Microprocessor group. Figure 6 illustrates the hardware architecture of JEM2.

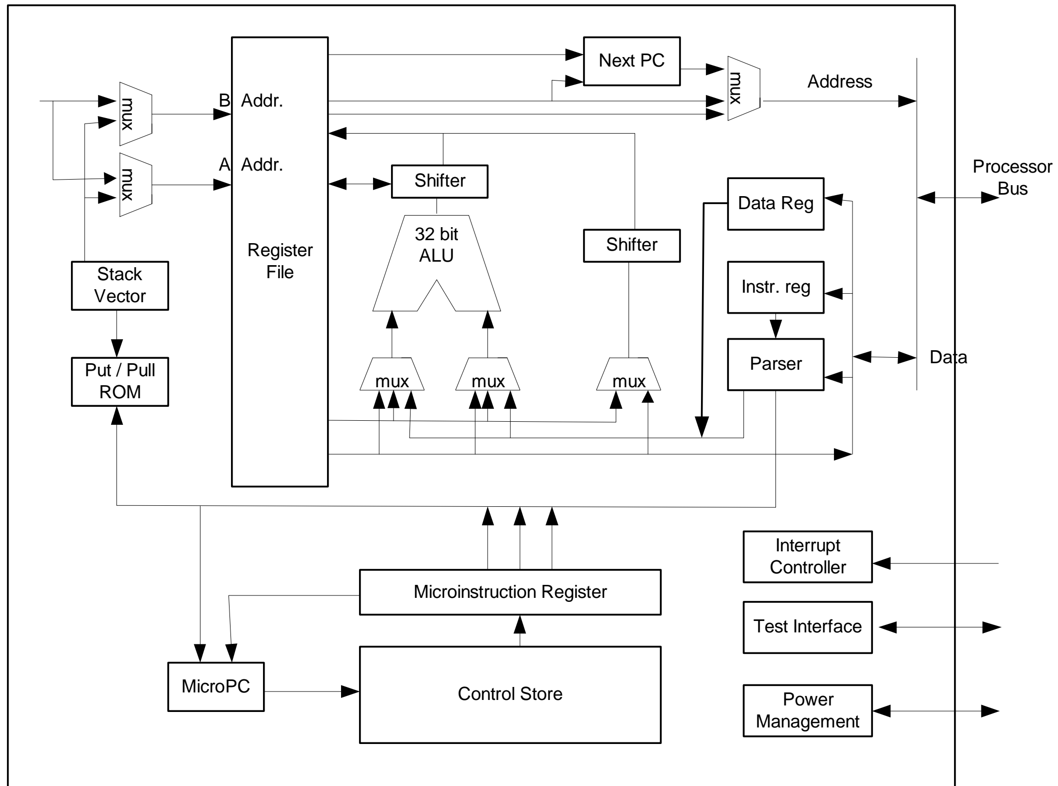

Figure 6 JEM2 Architecture, taken from [HAR].

Similar to LavaCore, JEM2 employs a register file that is used to implement the Java stack. Support is provided for a range of bus sizes and the Joint Test Action Group (JTAG) test and debug interface.

It is claimed that JEM2 implements the entire JVM instruction set in silicon and the only two bytecodes that trap immediately to software are *multinewarray* and *athrow* [HAR]. However, it can be seen in Figure 6, that the JEM2 core includes a “MicroPC”, a “MicroInstruction Register” and a “Control Store”. Although there are no more details in the public domain, it appears that JBCs are executed using micro-code and are not directly executed in hardware.

Certain instructions can be customised or new instructions can be added. An area of RAM is used to contain micro-code for custom instructions. Instead of requiring an assembler to gain access to the internal registers, a romizer substitutes method invocations with internal bytecode instructions. This feature is likely to make runtime development easier given the fact that low-level runtime routines can be developed using a standard Java compiler. Custom instructions can be defined in micro-code RAM using a configuration tool. Programmer defined static method names can then be mapped onto the custom instructions. They can then be used in programs by invoking the appropriate static methods. The romizer tool is also capable of *pruning* Java classes. It can detect unused classes, interfaces, methods, fields and constants. This can be used to reduce the memory requirements of the application and the runtime system. More interestingly it could also be used to remove dead code, as described in Section 2.4.

JEM2 provides micro-code support for real-time Java threads in order to improve performance. This includes threading features such as synchronization and scheduling. Hardware support is also provided for periodic threads and priority inversion control.

3.2.6 Choice of Hardware JVM

It was decided that a hardware JVM implementation approach would be used for the HRT JVM (see Section 3.2.5). However, the hardware JVMs discussed above have not considered non-determinism or been designed to support HRTSs (see Section 1.6). Therefore, they cannot be used as a starting point for this work. Furthermore, the above hardware JVMs are proprietary and so cannot be modified. Instead of developing a hardware JVM from scratch, the task was simplified by the discovery of the simple, public domain JVM discussed in [CLA97]. This machine was developed in the hardware description language Handel-C [CEL02], and could therefore be implemented on a field programmable gate array (FPGA).

The main benefit of the Oxford JVM was its simple timing model via the use of the Handel-C language and compiler (see Section 4.2 for discussion of Handel-C). The JVM implements a number of the simple JBCs, which are listed in Appendix 1.

However, the Oxford JVM falls short in a number of areas. It is only able to execute a sequence of very simple bytecodes. Methods cannot be called or returned from, and data cannot be stored in global variables. Currently all code and data must be held in one method of one class. Hence it is unable to support procedural or modular programming. There is no support for OO programming via Java objects. It is also unable to support threading. Finally the JVM cannot communicate with real-world devices, as there is no way for it to communicate with an external system bus. Despite these shortcomings, the Oxford JVM could be used. The next Section discusses the work required in order to implement the HRT JVM using the Oxford JVM as a starting point.

3.3 Work Required

In order to implement a HRT JVM using the Oxford JVM as a starting point, two phases of development work were required. These phases were split into preliminary work and additional work.

3.3.1 Preliminary Work

Before extending the Oxford JVM a number of modifications were necessary in order that it could be used as reliable staring point. These involved addressing a number of the low level problems.

A unit test suite was required in order to test each instruction. This meant that each instruction could be validated in order to ensure that it worked correctly. In order that an instruction test set could be constructed, a JBC assembler was required. Instead of developing a Java assembler from scratch, the Jasmin assembler was used [MAY97]. In order that internal JVM registers could be accessed during unit testing, a number of implementation specific instructions were added to Jasmin. These are discussed in Section 4.5.1.

It was not practical to use the Java compiler, as control over which JBCs were output for each test was needed. Bugs found during instruction testing were fixed in order that a reliable starting point was created. A number of other miscellaneous modifications were required, for example re-writing the main interpreter loop so that it was easier to

read and debug. The version of the Oxford JVM with the preliminary work carried out is referred to as the *Base JVM*.

3.3.2 Additional Work

As discussed in Section 3.2.6, the Oxford JVM is capable of executing only very simple instruction streams. In order to develop it into a form suitable to support HRT programming in Java, many additional features are required. These include support for complex JBCs, for example OO instructions. Given that there was a large shortfall between the Oxford JVM and a JVM to support HRT development, it was necessary to prioritise the individual features. Then by choosing an appropriate set of high priority features it was possible to provide a meaningful contribution towards the development of a HRT JVM. Table 1 lists the additional features required in order of priority.

Feature
1. Java class file assembler
2. Test suite for bytecode testing
3. Tidy up code
4. Debug processor
5. Romizer and Boot Loader
6. Support for procedural programming
7. Raw memory access, external bus and timer interface.
8. FPGA synthesis and test
9. Remaining simple bytecodes
10. Support for interrupt service routines and interrupt controller interface.
11. Object oriented bytecodes
12. Support for threads and synchronisation
13. Priority ceiling support
14. Threading classes
15. HRT Class library

Table 1 Feature prioritisation.

As discussed earlier, in order to create a reliable base JVM, a number of preliminary tasks were completed first. Without a class file assembler the base JVM could not be validated as a reliable and robust starting point, hence this was done first.

The test suite used to validate the correct operation of each bytecode was necessary once the assembler was available and hence given a priority of 2. Other preliminary

work included tidying up the source code and making it easier to read and debug. This was done after the assembler and test set had been developed. The processor was debugged, after which point the Oxford JVM became a reliable and robust starting point referred to as the *Base JVM*.

In order that compiled class files could be executed by the JVM a *romizer* and *boot loader* were required. See Section 4.5 for a more detailed discussion of the romizer and bootloader. These were assigned a priority of 5, as without them, an application cannot be executed.

Perhaps the most serious deficiency of the Oxford JVM is the lack of support for procedural programming. As procedural programming is a fundamental requirement of any procedural or OO language, it was given a priority of 6.

In Section 1.2 it was stated that HRTSs must monitor and control several different aspects of the environment. Hence raw memory access, external bus and timer support were given a priority of 7. Without these features the JVM cannot communicate with the outside world. Following this, is hardware synthesis in order that the JVM can be deployed. It can be seen that the features prioritised up-to this point mean that a practical HRTSs can be supported.

With a priority of 9 is the implementation of the remaining simple (primitive stack based) JBCs. Interrupt service routines (ISRs) are required in order to allow asynchronous external events to be handled in a multi-threading environment. Support for ISRs was given a priority of 10. OO instructions are fundamental to Java. Section 2.4, highlighted the fact that there are potential problems surrounding the use of object orientation in HRTSs. However, the author believes that there is justification for supporting a simplified and carefully implemented object-based system. As such, object oriented support is given a priority of 11.

On-line scheduling for HRTSs is becoming more common in order to reduce the costs and difficulties in constructing and maintaining cyclic executives [BUR97] and as such

support for threading is given a priority of 12. Synchronisation must be supported for safe data and resource sharing [BUR97], and was also given a priority of 12. However in order to ensure that any application is free from priority inversion [BRI99], priority inversion protection is given a priority of 13. A simple threading class library, based on the RTSJ, is required in order that the programmer can interface to the scheduler. This was given a priority of 14.

Finally, in order that the programmer can be productive using the JVM a small set of HRT class libraries is required, carefully designed and programmed to be deterministic and analysable for WCET. This was given a priority of 15.

It was unrealistic to consider implementing all of the above features due to the large amount of effort required and limited time available. However it was considered that the features up to and including support for procedural programming (priority number 6) could be achieved. Providing support for procedural programming would mean that the most serious deficiency of the Oxford JVM would be addressed. Hence it could be argued that a meaningful step towards implementing a hardware JVM for HRTSs could be achieved.

The work completed for priority 1 involved minor modifications to an existing Java class file assembler. The features listed as priorities 2 – 6 were developed in full as part of this work

3.4 Summary

There are a number of approaches to Java execution. These include interpreter, JIT, ahead of time compilation, hardware acceleration and hardware JVM. It is argued that the least non-deterministic option that allows for the greatest control during implementation is the hardware JVM approach. Hence it was decided that the hardware JVM approach would be used for the HRT JVM.

The development of a *base JVM* was used as a starting point for the development of a HRT JVM. This was simplified by the discovery of a simple, public domain hardware

JVM discussed in [CLA97]. A set of tasks was identified in order to provide a complete implementation of a JVM for HRT.

Providing support for procedural programming would mean that the most serious deficiency of the Oxford JVM would be addressed and a meaningful step towards the implementation of a hardware JVM for HRTSs could be achieved.

4 The Development of JakHarta: a JVM to Support HRT Systems

4.1 Introduction

This Section describes the design and implementation of a HRT JVM using the Base JVM as a starting point. For the sake of convenience the name given to the HRT JVM is *JakHarta*. Background information is presented describing the Java execution stack and class file format before considering the general behaviour of newly added instructions. The design of each new instruction and supporting runtime data structures is presented with particular emphasis on constant time execution. The implementation aspects are then discussed.

4.2 Development Environment

As the starting point for the practical work is the Oxford JVM, this means that Handel-C will be used for the implementation. Handel-C [CEL03] is a synchronous programming language based on ANSI C, but modified to allow behavioural hardware synthesis. A simple timing model is used at the language level, which states that each assignment will always take exactly one clock cycle. From this rule it is very easy to determine cycle times for implementations. Parallelism can be explicitly defined using a C like syntax but with Occam [INM88] semantics. A software simulation can be developed initially which can be further developed into synthesizable form for an FPGA. The DK2 development environment [CEL02] provides a graphical user interface language editor, project-build system and debugger.

4.3 Support for Procedural Programming

As stated in Section 3.2.6, a major limitation of the Oxford JVM is the lack of support for procedural programming. Once preliminary work was done to create the base JVM, the first meaningful step towards the goal of a HRT JVM was to extend it to support procedural programming. In the simplest form this means supporting static methods and static class variables. Both belong to a class, rather than specific instances. Hence early binding is used for static methods, and so they can be implemented separately from the OO instructions. Static methods are similar to function calls in C. Hence by

implementing static methods and variables, simple HRT programs can be written in a procedural fashion. Such programs have multiple classes, each having only static methods and variables, and one is the 'main class' which contains *public static void main*. In such programs static class variables and methods play the role of global variables and functions in procedural programs.

```
1       public class Caller
2       {
3             public static int result;
4
5              public static void main()
6              {
7                  demo();
8              }
9
10            public static void demo()
11             {
12                      int local1 = 1;
13                      int local2 = 2;
14
15                      Callee.calc(local1, local2, 3);
16             }
17      }
18
19      public class Callee
21
20      {
22            public static void calc(int arg1, int arg2, int arg3)
23             {
24                      int local1 = arg1;
25                      int local2 = arg2;
26                      int local3 = arg3;
27
28                      Caller.result += (local1 * local2 + local3);
29             }
30      }
```

Figure 7 Procedural Java program using static methods and variables.

Figure 7 shows a procedural Java program using static methods and variables. There are two classes, *Caller* and *Callee*. On line 3, the static class variable *result* is defined, which belongs to the class *Caller*. This is then manipulated on line 28. Lines 12-13 and 24-26 use local variables and method parameters. Lines 7 and 15 invoke static methods.

As well as using procedures, it can be seen that the program is made up of more than one class where each class could potentially contain a number of static class variables and associated methods. Therefore program state and behaviour can be wrapped in a cohesive compilation unit, which supports encapsulation. Static methods can contain local variables, which allow automatic scoping to be used. The Java language features used in this example are comparable to global functions and variables in C. Such features allow simple HRT programs to be supported.

Figure 8 shows the Java assembly bytecode for the *demo* and *calc* methods in Figure 7, generated by the Java compiler.

```
1     public static void demo()
2     {
3        iconst_1
4        istore_0
5        iconst_2
6        istore_1
7        iload_0
8        iload_1
9        iconst_3
10       invokestatic 21 (com/HRT_Java/RTDS/Callee.calc)
11       ret
12    }
13
14     public static void calc(int arg1, int arg2, int arg3)
15     {
16        iload_0
17        istore_3
18        iload_1
19        istore 4
20        iload_2
21        istore 5
22        getstatic 15 (com/HRT_Java/RTDS/Caller.result)
23        iload_3
24        iload 4
25        imul
26        iload 5
27        iadd
28        iadd
29        putstatic 15 (com/HRT_Java/RTDS/Caller.result)
30        ret
31    }
```

Figure 8 Java assembly bytecode for the demo and calc methods.

It can be seen that when the Java compiler encounters a static method call, for example *Callee.calc(local1, local2, 3)* on line 15 of Figure 7, it generates an *invokestatic* instruction as can be seen on line 10 of Figure 8. When the Java compiler encounters a

static field read/write as in line 28 of the Java program, it generates a *getstatic* or *putstatic* JBC respectively. This can be seen on lines 22 and 29 of Figure 8.

In order for JakHarta to support procedural programming, five Java bytecodes (JBCs) have been added to the base JVM. These are *getstatic*, *putstatic*, *invokestatic, return* and *ireturn*. *getstatic* and *putstatic* are used to read and write static class variables. *invokestatic, return* and *ireturn* are used to invoke and return from static methods. Before discussing the design and implementation of these new JBCs, fundamental aspects of the JVM must be first discussed.

4.3.1 The Java Virtual Machine

In order to implement the JVM, elements that must be considered are the internal registers, execution stack, and class file. The internal registers are platform specific and not defined by the JVM specification [LIN99]. For example, a JVM may define a set of internal registers to store the execution state of a program. At the basic level these registers may include a *program counter* (PC) and an *instruction register* (IR). When executing a program, the JVM fetches the next JBC in memory referred to by the PC. The JBC is then stored in the IR. Depending on its value, the appropriate routine is invoked in order to execute the required behaviour of the JBC.

The execution of a Java program makes use of a stack as shown in Figure 9. The stack is used to hold operands, and method *activation records*. When a method (either static or non-static) is called, an activation record is pushed onto the stack. This is the data space of the method and provides storage for the method's parameters and local variables. The parameter and local variable sections are organised together as an array. In Figure 9, a *Stack Base r*egister (SB) is used to refer to the base of this array. Load and store instructions that manipulate method parameter and local variable sections provide an index into this array when referencing these items. The index is added to the SB in order access the required parameter or local variable. The frame data (see Figure 9) is platform specific and includes control flow information such as the return address, saved register values and a reference to the *constant pool* (see below) for the current method's class file. Finally the operand stack is used for storing the results of

expressions evaluated in the body of the method. The SP register is used to refer to the top item on the operand stack. Once the activation record has been created, a jump is made to the start of the bytecode, which implements the method body. When the method returns, its activation record is removed from the stack. Figure 9 illustrates this.

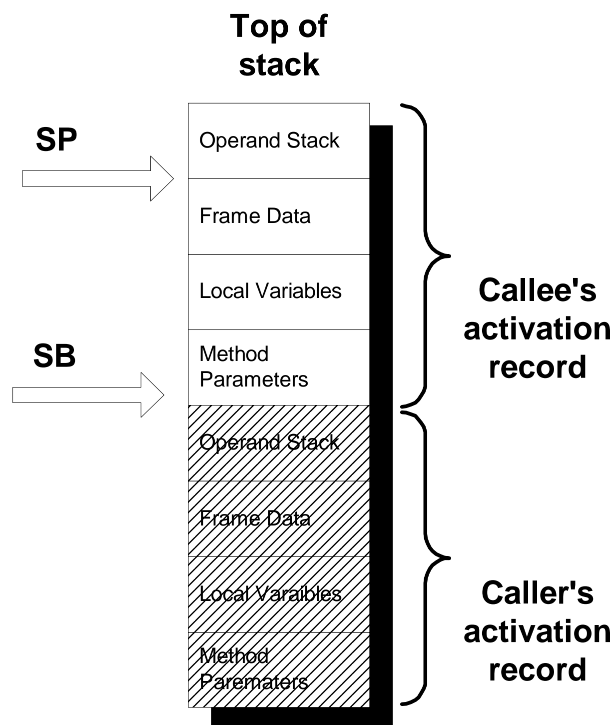

Figure 9 JVM stack showing 2 activation records.

The stack word size must be large enough to hold values of types byte, short, int, char, float and reference [LIN99]. A requirement of the JVM specification is that two words must be large enough to hold a value of type long or double [LIN99].

A class in a Java program is compiled to a *class file*. This contains information required by a JVM to execute instances of the class. The contents of a class file include information about the attributes of the class, method tables which store the JBC for the methods of the class, and a *constant pool*. The constant pool is necessary in order to aid dynamic loading and linkage, for example via the Internet. The constant pool is a symbol table, whose entries include information about classes, instance variables and

methods. This information is used to search the method and fields tables for the required bytecode or variable details. Each entry has a number of fields, which contain string and numeric literals as opposed to platform specific memory address offsets. A number of JBCs reference the constant pool entries via an index, for example when invoking a method and reading or writing a static class variable. This is opposed to referencing the instructions and data directly as in compiled C code. Because constant pool entries are symbolic, they are platform independent and make no assumptions of the underlying JVM or hardware platform. This means that compiled class files can be loaded and executed by any compliant JVM, regardless of how it represents or translates classes internally.

4.3.2 Behaviour of the invokestatic return and ireturn Bytecodes

As described above, when a static method is called, for example Callee.calc(local1, local2, 3), the Java compiler generates an *invokestatic* bytecode, for example, *invokestatic 21*. The argument 21 is an indirect reference to the location of the method's bytecode. Prior to the *invokestatic* instruction, method parameters are pushed onto the stack (as seen in lines 3 to 9 in Figure 8), and the method's activation frame is then set up as a consequence of the *invokestatic* instruction. *invokestatic* needs certain information from the called method's class file to create the activation record, including the location of the method's bytecode, the number of parameters and the number of local variables. The location of the constant pool for the class *that is invoking* the method is also required since the above information is located in the *caller's* constant pool. The argument of *invokestatic* is used at runtime as an index into the constant pool of the *caller's* class file. The target constant pool entry is a data structure of type *CONSTANT_Methodref_info* which provides information about the method being called, including its name and signature. The signature of a method is a textual description of the parameters and return type information encoded as a string, for example, (*III)V* means that the method takes 3 integer parameters and returns void. This information enables the method to be found in the method table section of the class file.

When a *return* or *ireturn* instruction is encountered the method activation record is destroyed and the activation record for the calling method is restored. In the case of an

ireturn the integer return value resulting from the method invocation is left on top of the *Caller's* expression stack.

4.3.3 Behaviour of the getstatic and putstatic Bytecodes

When a static class variable is read or written to for example, Caller.result in Figure 7, the Java compiler generates a *getstatic* or *putstatic* bytecode respectively, for example, *getstatic* 15 or *putstatic* 15. The argument 15 is an indirect reference to the location of the static class variable. *getstatic* reads the value of the static class variable from memory and pushes the value onto the operand stack. *putstatic* pops the value off the operand stack and writes the value to the memory address of the static class variable. Hence, both instructions need the location of the static class variable. The argument of *getstatic* and *putstatic* is used at runtime as an index into the constant pool of the class that is accessing the static class variable. The target constant pool entry is of type CONSTANT_Fieldref_info and provides information about the static class variable being accessed, including its name and type. This information enables the static class variable to be found in the field table section of the class file.

4.4 Constant Pool Resolution

As discussed in the previous section, the argument of *invokestatic*, *getstatic* and *putstatic* is used at runtime as an index to an entry in the constant pool of the class that is *calling* the static method or accessing the static class variable. The target constant pool entry provides information about the method being called or static class variable being accessed, including the name and type.

In order to support platform independence and dynamic class loading in a desktop environment, the constant pool entries relating to instruction parameters are symbolic and cannot be used directly by a JVM implementation. The information must be processed first. Hence, symbolic constant pool entries must be *resolved* at run-time. The constant pool entries need to be replaced by platform specific entries that include the actual address of target method's bytecode or static class variable. In a desktop JVM, the resolution is performed at runtime by the dynamic class loader.

As discussed in Section 2.4, Java class loading is highly non-deterministic and is both unnecessary and unsuitable for HRTSs. Because of this JakHarta does not use dynamic class loading. Instead, it requires that resolution be performed ahead of time on the host development system, using a *romizer*. The romizer and other aspects of the JakHarta tool chain are discussed in the next section before describing the design of the runtime constant pool.

4.5 JakHarta Tool Chain

In order to execute a HRT Java program on JakHarta, an executable memory image must be generated. This is achieved via the JakHarta tool-chain running on the host development PC, which consists of a Java assembler, Java compiler, WinZip and *Romizer*. Figure 10 illustrates the data flow for the tool chain.

Figure 10 JakHartaTool chain data flow diagram.

4.5.1 Java Assembler

Because standard Java compilers do not allow the JVM internal registers or data structures to be exposed, this means that Java cannot be used for low-level runtime system development. A Java assembler is used to provide low-level access to JakHarta registers such as program counter (PC), stack base (SB), stack pointer (SP), and return stack pointer (RSP) during unit testing. JakHarta registers are discussed further in Section 4.7.

Instead of developing a Java assembler from scratch, the Jasmin assembler was modified and used instead. Jasmin is a free Java assembler available from [MAY97]. It takes as input a textual description of a Java class using the JBCs and translates this in to a binary class file. In order to provide access to JakHarta registers in the assembly language unit tests, a number of implementation specific instructions have been added to Jasmin. Table 2 lists the additional instructions supported by the modified Jasmin assembler.

Instruction	Purpose
pushSp	Pushes SP onto the data stack.
pushSb	Pushes SB onto the data stack
pushRsp	Pushes RSP onto the data stack.
pushPc	Pushes PC onto the data stack.
popSp	Pops the top data stack word into SP.
popSb	Pops the top data stack word into SB.
popRsp	Pops the top data stack word into the RSP.
popPc	Pops the top data stack word into the PC.

Table 2 New JakHarta instructions added to Jasmin assembler.

4.5.2 Java Compiler

A standard Java compiler is used as the main source code compilation tool. This takes a number of Java language source files and translates them in to standard Java class files. The generated class files are then processed by the archiving tool and used by the romizer in order to generate an executable image for JakHarta.

4.5.3 WinZip

It is conventional to collate and compress a number of related Java class files into a single file, called a Java Archive (JAR) file. WinZip is used for this purpose. The romizer decompresses the JAR files in order to extract the classes.

*4.5.4 **Romizer***

Dynamic class loading is inappropriate for HRTSs (see Section 2.4), and is not supported by JakHarta. Classes must therefore be translated into a format suitable for execution at compile time. In essence, a *romizer* is a host-based software tool that extracts information from a set of Java class files and creates a corresponding set of run-time data-structures (RTDS) to enable instances of the classes to be executed by the JVM.

The romizer applies a number of processes to each of the class files in order to produce an executable JakHarta memory image. Figure 11 illustrates this.

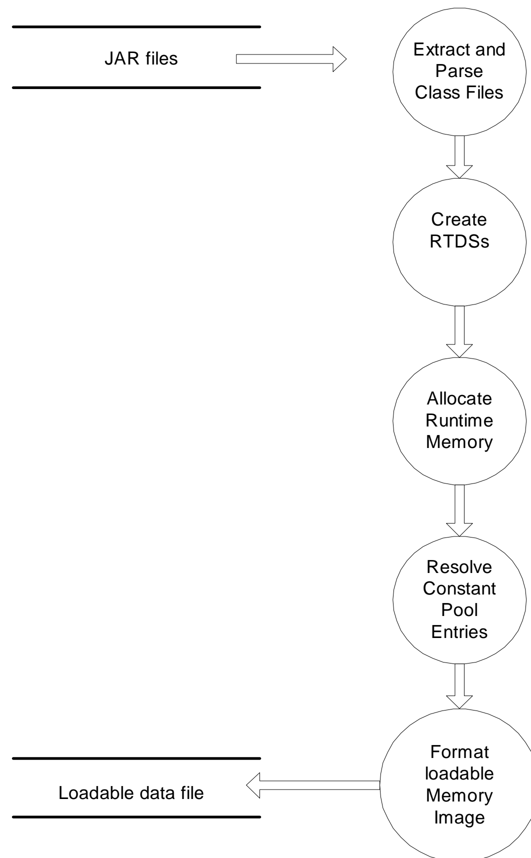

Figure 11 Romizer Data Flow.

The first process of romizing is to extract the class files from the JAR files and parse them into an intermediate representation. This allows the romizer to easily search and retrieve class file information during later processing.

JakHarta RTDSs are then created for each class. These represent the runtime constant pool, method bytecode and static class variables (discussed in Section 4.3.1). Some of the RTDSs can be populated with the necessary information at this point, for example the RTDSs representing the method bytecode can be populated. However, runtime constant pool entries cannot be generated until all RTDSs have been assigned runtime memory addresses.

The next process is to assign the RTDSs with runtime memory addresses according to the memory layout policy (shown in Figure 12). The assigned runtime memory addresses determine where the runtime data structures are placed in memory during the boot-up procedure.

Once all the RTDSs have been assigned memory locations, the romizer *resolves* the constant pool entries that represent static methods and variables. This process may otherwise be referred to as *linking*. The format of the runtime constant pool created by the romizer and used by JakHarta is discussed below in Section 4.6.

The final process is to output the RTDSs to a data file in a format suitable for loading into memory during boot-up.

4.6 Run-time Constant Pool for JakHarta

The primary aim for JakHarta is that all instructions should execute deterministically and efficiently so that it can be used for HRTSs. This includes those instructions that access the constant pool such as the *invokestatic*, *getstatic* and *putstatic* bytecodes. Therefore a design constraint for the runtime constant pool is to achieve constant time access.

Figure 12 illustrates the runtime data structures representing the runtime constant pools generated by the romizer. This is an array of constant pools; there is one constant pool for each class in the program. Within each pool there is an entry for each method called or static class variable accessed by the code of the class.

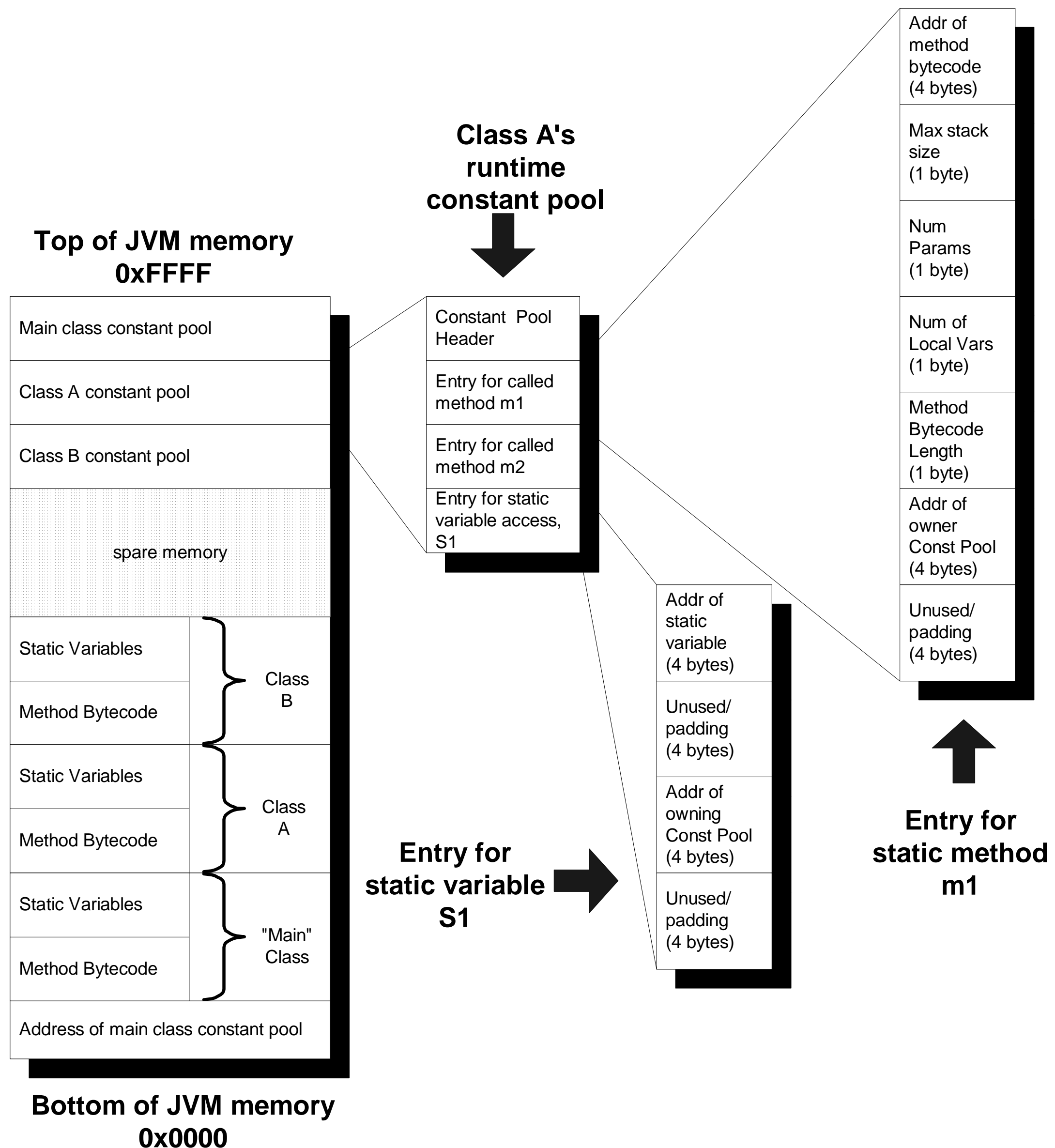

Figure 12 JakHarta Runtime constant pool and memory layout.

At the start of each constant pool is a header. Being the first constant pool entry, this is unused by the Java compiler. The runtime constant pool data structures make use of this entry to store the number of entries that the constant pool contains.

The static method entries are used by the *invokestatic* instruction and relate to *CONSTANT_Methodref_info* entries in the Java class file (see Section 4.3.2). The address of the bytecode refers to the address of the *called* method's bytecode and is

generated by the romizer (see Section 4.5.4). The maximum stack size originates from an entry in the original class file, and includes space for parameters, local variables and expressions. It is currently used for debugging purposes to indicate if the stack can overflow whilst executing the method. The number of parameters and the number of local variables also originate from the class file, as does the number of bytes in the bytecode stream. The address of the constant pool is the address corresponding to the class that owns the *called* method and is generated by the romizer.

The static field entries are used by the *getstatic* and *putstatic* instructions and relate to *CONSTANT_Fieldref_info* entries in the class file (see Section 4.3.3). The address of the static field points to the static class variable data in memory. The address of the owning constant pool is the address corresponding to the class that owns the field. These are both generated by the romizer.

Access to runtime constant pool entries has been made constant time by making all entries a uniform size of 16 bytes. For example, a constant pool entry for a static class variable only actually requires that 8 bytes of information is stored (see Figure 12). However 8 bytes of additional padding are included in order to make it the same size as a static method entry. Access to a constant pool entry is found as follows: first, the index from *invokestatic*, *getstatic* or *putstatic* is multiplied by the uniform size of a constant pool entry, 16 bytes (a shift left of 4 places is used). This is then added to the starting address of the constant pool, which is stored on the return stack (see Figure 13). Hence it always takes the same number of clock cycles to locate a runtime constant pool entry, regardless of its type or position within the constant pool.

This approach trades increased constant pool memory requirements for constant access time. It is possible to allow constant pool size to vary, and to compact constant pools, for example, by removing unused entries. However, such memory saving schemes are likely to require that the constant pool entries be linked together in order that they can be sequentially *searched* for the required entry instead of being able to simply calculate the position of an entry directly. Hence rather than enabling access to an entry in

constant time, as in the current approach, the WCET to access an entry would be proportional to the number of constant pool array entries.

Figure 12 shows the JakHarta memory map and indicates how the runtime data structures are laid out in memory. The constant pools are located at the top of memory, and method bytecode and static class variables are at the bottom. By separating constant pools and methods in the address space, it means they can be located quickly during debugging. *main's* bytecode is always located at memory location 4 as this is the fixed location which JakHarta begins executing during boot-up. Other bytecode streams belonging to the main class are located sequentially in bottom-up order after *main*. Then all static class variables belonging to the main class are located sequentially in a bottom-up order. The bytecodes and static class variables of other classes are organised in a similar manner.

4.7 New Instruction Design

4.7.1 invokestatic

A JVM implementation may decide to store method activation records on the heap rather than the stack. However, since the base JVM has no heap, the simpler task of implementing a stack-based scheme was preferred in this phase of the work. An additional consideration is that it is more straightforward to optimise the parameter-passing process in the context of a stack (see below).

JakHarta uses a separate return stack for the *invokestatic* and *return* bytecodes. This means that flow control information relating to method invocations is kept separate from program data. This aided a simpler design and also allowed for easier debugging during the implementation.

Providing a program is written using HRT idioms and can be analysed for worst-case stack usage, the stacks can in principle be made big enough to ensure an overflow can never occur. However, as suggested in Section 2.7, stack depth analysis is not possible in the Puschner HRT profile since the initialisation phase is unpredictable. In this case,

automatic stack spilling would be necessary when the JVM encounters an overflow or underflow during the execution of an *invokestatic* or *return*. As well as making a system unpredictable it is clear that the Pushner profile would further complicate the JVM and WCET analysis. This further emphasises the need for a deterministic initialisation phase as well as execution phase.

When a method is called, activation records are set up on both stacks, and when it returns, both are discarded. The data stack holds parameters, local variables and the expression stack, whilst the return stack contains control information that enables the calling method to be resumed when the called method returns.

Two registers, stack base (SB) and stack pointer (SP) contain the addresses of the bottom and top of the *callee's* data stack activation record. SB is used to point to the first method parameter or first local variable and SP points to the last local variable or last expression result. SB is used by *iload* and *istore* bytecodes which are used to access method parameters and local variables. When generating these bytecodes, the Java compiler specifies an offset into the zero-based array that contains the method parameters and local variables. SB is therefore used as the base address of the array so that when added to the method parameter or local variable offset, the required stack element is accessed. Should no method parameters or local variables exist for the method, SB is unused and the method bytecode will not contain any *iload* or *istore* bytecodes.

The information stored on the return stack includes the *caller's* SB and SP. The return and the constant pool addresses are also stored. The return address is the address of the bytecode in the calling method immediately after the *invokestatic* bytecode and the constant pool address is the location of the of the *callee's* constant pool in memory. Figure 13 illustrates the data and return stacks for the method invocation on line 10 of Figure 8.

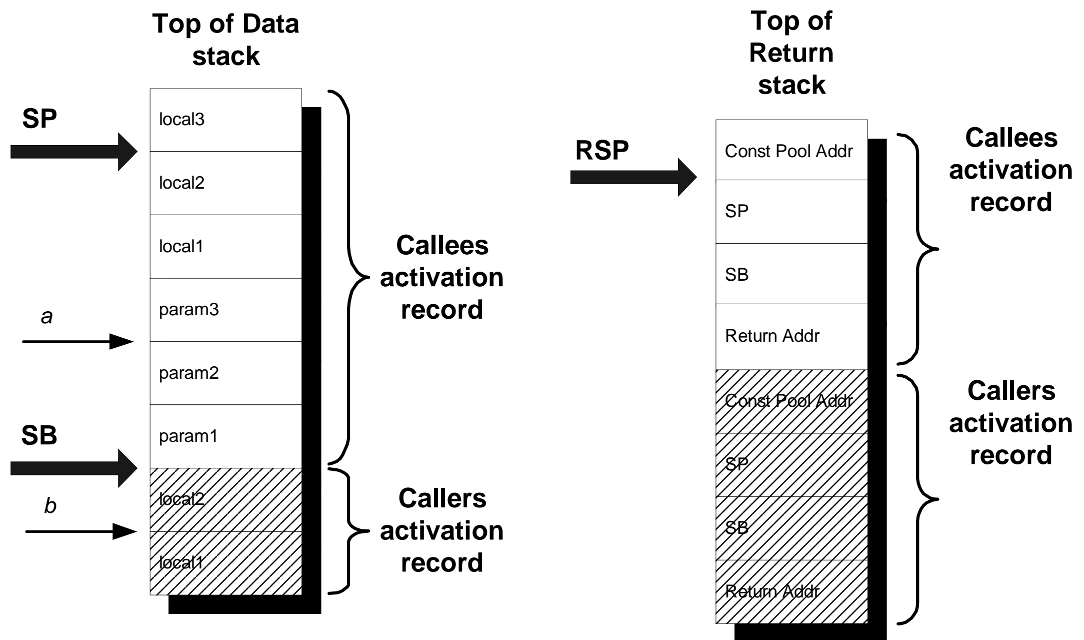

Figure 13 Data and return stacks for a method invocation.

The *invokestatic* bytecode on line 10, Figure 8, has a parameter that references the *caller's* constant pool entry for the *called* method. The information in this entry enables the data stack activation record for the called method to be constructed. However, before this is done, the return stack activation record is set up by saving the return address, SB and SP. This can be seen in the *callee's* activation record on the return stack in Figure 13. Note that SP is modified before being pushed, for reasons discussed below. If a method has *N* arguments, then the Java compiler plants code to push them on to the stack prior to the *invokestatic* bytecode. At this point SP would point to *a* in Figure 13. If SP is restored to this value when a *return* JBC is executed, the method parameters would still be live on the data stack. Hence a simple, familiar optimisation is to reset SP to point to *b* in Figure 13 prior to pushing it on to the return stack so that when the call returns the parameters are discarded. After the modified stack pointer value has been pushed on to the return stack, the address of the *called* method's constant pool is pushed. The top of the return stack pointed to by the return stack pointer (RSP), as seen in Figure 13, therefore contains the current constant pool address. This is used to access the *callee's* constant pool when executing its method bytecode.

Finally, to complete the static method invocation, the JVM jumps to the start of the bytecode for the called method, via the address in the constant pool entry (see Section 4.6).

4.7.2 return and ireturn

At some point during the execution of the called method's bytecode, a *return* instruction may be encountered by the JVM. This indicates that control needs to pass back to the caller, more specifically to the instruction after the *invokestatic* in the *caller's* method bytecode. Before transferring control back to the caller, the previous activation records need to be restored.

The data stack for the caller is restored first using the information contained on the return stack. The current constant pool address is discarded and the SP value is set which is obtained by popping it off the return stack. For an *ireturn* bytecode, the top of the *callee's* data stack is copied to the top of *caller's* expression stack. This is shown in Figure 14 as SP'.

The stack base pointer for the caller is then restored, by popping it off the return stack. Figure 14 illustrates the data stack and return stack in their restored states.

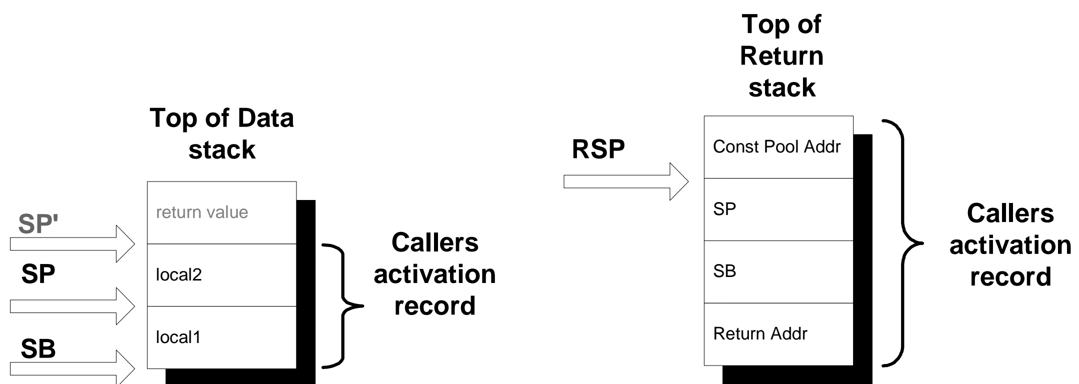

Figure 14 Data and Return stacks after a return/ireturn.

Now a jump needs to be performed to the instruction after the *invokestatic* in the *caller's* bytecode. The address of this instruction, which is the return address, is

obtained by popping the return stack and writing the value to the program counter (PC). Notice that by popping the return address off the return stack, the return stack pointer is left correctly pointing to the constant pool address for the *caller*. This allows future references to the *caller's* constant pool to be accessed without further manipulations.

4.7.3 getstatic and putstatic

getstatic and *putstatic*, like *invokestatic*, have parameters which refer to the constant pool of the class which is accessing the static field. The information in this entry includes the address of the static class variable in memory.

The *getstatic* instruction simply reads the value of the static class variable at the memory location specified by the constant pool entry and pushes it on to the data stack. The *putstatic* instruction pops the value off the top of the data stack and writes it to the memory address for the static class variable.

4.8 Implementation of New Instructions

The *getstatic*, *putstatic*, *invokestatic, return* and *ireturn* JBCs were implemented in Handel-C as extensions to the base JVM. Extensive simulations have been run, including recursive Java programs (see Section 5.4). However, no synthesis has yet been attempted. In the simulations, the data and return runtime stacks were located on-chip in order to improve performance of primitive stack operations and also more complex instructions responsible for method invocation and return.

Both stacks are currently set to 32 words, however, because HRT programs are required to be statically analysable, it is thought reasonable that the sizes of the stacks can be set depending on the worst-case stack usage of the application. Alternatively assuming that all aspects of the HRT program are statically analysable, the point at which any stack overflow or underflow occurs can be determined and handled by the application.

The Handel-C simulator provides gate-count estimates, and the new instructions led to a 1.6% increase in gate count, and a 17% increase in flip-flop count over the base JVM. Note that in the base JVM, the *imul* instruction accounts for 25% of the gates [GOB04].

The number of clock cycles needed to execute each of the above instructions is constant, as indicated in Tables 3 and 4 below. The *activity* executed in each cycle for each instruction is shown in Table 3. Some *activities* require more than one clock cycle. For example, the action "get the index of the called method's entry in the *caller's* CP" for the *invokestatic* instruction, starts at clock cycle 2 and continues into clock cycle number 3. An arrow is used to show this. In this example, the Handel-C implementation of the activity requires 2 assignments. As discussed in Section 4.2, the timing model of Handel-C states that each assignment statement takes 1 clock cycle.

Clock Cycle	**invokestatic**	**return/ ireturn**	**getstatic**	**putstatic**
1	Get *caller's* constant pool (CP) address. This is on top of the return stack (RS), pointed to by RSP.	Pop saved SP value from RS. ↓	Get *caller's* constant pool (CP) address. This is on top of the return stack (RS), pointed to by RSP.	Get *caller's* constant pool (CP) address. This is on top of the return stack (RS), pointed to by RSP.
2	Get the index of the *called* method's entry in the *caller's* CP. This is a parameter of the *invokestatic* bytecode. ↓	↓	Get the index of the static class variable's entry in the *caller's* CP. This is a parameter of the *getstatic* bytecode.	Get the index of the static class variable's entry in the *caller's* CP. This is a parameter of the *getstatic* bytecode.
3	↓	↓	Calculate the offset of the static class variable's CP entry by multiplying the above index by 16.	Calculate the offset of the static class variable's CP entry by multiplying the above index by 16.
4	Calculate the offset of the method CP entry by multiplying the above index by 16.	Pop saved SB value from RS. ↓	Add offset to the callers CP address in order to locate the static class variable's CP entry.	Add offset to the callers CP address in order to locate the static class variable's CP entry.
5	Add the above offset to the callers CP address in order to locate the method CP entry.	↓	Get *Addr of static field* from the CP entry.	Get *Addr of static field* from CP entry.

6	Now create the RS activation record. To start with RSP is incremented so that the *caller's* activation record is not overwritten.	Pop saved Program Counter value from RS.	Read the value of the static class variable from memory.	Pop value from DS.
7	Push the return address to the RS.		Push the value of the static class variable to the DS.	
8				Write to memory location of static class variable.
9	Push SB to the RS.		Set the Program Counter to the next bytecode in bytecode stream.	
10				
11	Get the number of method parameters for the called method. This is obtained from the *Num Params* field in the *called* method's CP entry.			
12	Reset SP by subtracting the number of parameters. This means that when SP is restored during the return, the method parameters are destroyed. Push SP to the Return Stack.			Set the Program Counter to the next bytecode in bytecode stream.
13				
14	Push the address of the called method's CP to the RS.			
15				
16				
17	Get *Num of Local Vars* from the *called* method's CP entry.			
18	Add the *Num of Local Vars* to SP, in order to create space for the local variables.			

19	Set SB to point to the first method parameter or first local variable. SB is then used as a base pointer for accessing method parameters and local variables during execution of the method's bytecode.			
20	↓			
21	Jump to the callee's bytecode stream by setting the program counter to *Addr of bytecode method* in the *called* method's CP entry.			

Table 3 Constant clock cycle times for new instructions.

JBC Instruction	Number of Clock Cycles
invokestatic	21
return/ ireturn	7
getstatic	9
putstatic	12

Table 4 Summary of cycle times for new instructions.

4.9 Summary

One of the biggest constraints of the base JVM is the lack of support for procedural programming, and so procedural programming was implemented in JakHarta via static methods and static class variables. These provide similar capabilities to functions and global variables in C.

The Java class file contains platform independent information for the class such as the method code and constant pool, however this is in symbolic format and is unsuitable for direct execution by a JVM implementation. A host-based romizer tool was developed along with modifications to a Java bytecode assembler. The romizer and assembler,

together with a standard Java compiler and class file archiving tool are known as the JakHarta tool chain, and are discussed in Section 4.5. The romizer extracts information from the Java class files and generates a set of loadable and executable runtime data structures specifically designed for constant time access. A data file is produced by the romizer, which represents the runtime memory image of the HRT application. This is read and laid out in JakHarta's memory address space at boot-up time.

In order to support procedural HRT programming, the Java instructions *invokestatic*, *return*, *ireturn*, *getstatic* and *putstatic* have been designed for constant-time execution.

A data stack was used to store method parameters, method variables and expression results. A separate return stack was used for holding method context information such as the current methods constant pool address, the *caller's* return address and the *caller's* data stack register values.

Extensive simulations confirm that due to the design and implementation of the new instructions and runtime data structures, they all execute in constant cycle time.

The inclusion of the new instructions resulted in a relatively small increase in FPGA gate count. This further implies the feasibility of implementing the more complex bytecodes in hardware instead of software in order to improve performance.

5 Test Results

5.1 Introduction

The design and implementation of the new instructions, which support procedural programming, were discussed in the previous chapter. In this chapter three sets of test results are presented. At the most basic level, unit tests were developed in order to validate the correct behaviour of each individual JBC instruction. Then the simple program, shown in Section 4.3, was used to demonstrate how the new instructions execute correctly as part of a real Java program. A more complex test was used to demonstrate the predictable and scalable execution time of a recursive Java program.

5.2 JBC Unit Tests

A number of extensive unit tests were developed to validate the correct operation of each individual JBC. Simple JBCs have only one or two tests, whereas complex JBCs, for example *invokestatic*, have a number of separate unit tests. In total these amounted to some 80 tests. The unit tests were also used for regression testing during development of JakHarta to ensure that bugs were not introduced as a result of a modification to the JVM. Java assembly language (using the Jasmin assembler, see Section 4.5.1) was used in order that the internal registers could be read, written and validated at certain points during test program execution. The *breakpoint* JBC was used to halt program execution and output test result information to the Handel-C debugger output window. Employing Java assembly language meant that the exact bytecode instructions could be generated for the unit test programs. This is instead of attempting to rely on the Java compiler to generate the required instructions.

The unit tests identified a number of bugs with the existing instructions. However, once these were fixed and the existing bytecodes had been validated as working correctly in accordance with the JVM specification [LIN99], a number of simple Java programs were developed for further validation. An example of a bytecode unit test written in

Java assembly language using the modified Jasmin assembler is shown in Figure 15[14]. The *ifeq* JBC is used to branch to a specified target if the top data stack word is equal to zero. If the value is non-zero, then execution continues immediately after the *ifeq* JBC.

```
1   .class public JVMTest5              ; this is the name of the test
2   .super java/lang/Object
3
4   .method public static Main()V     ; this is the name of the method, main()
5
6   .limit locals 0    ; used to tell Java verifier that there are no local variables
7
8   ;--------------------------------------------------------------------------
9   ;  Setup sb/sp as per number of main() method args and local vars.
10  ;  There are none for this test, so set both to zero.
11  ;--------------------------------------------------------------------------
12    bipush 0
13    popSb        ; internal JakHarta instruction used to write to sb register
14
15    bipush 0
16    popSp        ; internal JakHarta instruction used to write to sp register
17
18   ;--------------------------------------------------------------------
19   This is where the test for the ifeq bytecode is actually done
20   ;--------------------------------------------------------------------
21   iconst_0              ; push 0 on top of data stack
22   ifeq test5_ok         ; this should evaluate to true and execution should branch to target address
23
24   goto test5_fail
25
26   test5_ok:              ; the branch happened, which is the correct behaviour
27   iconst_1               ; now test that a branch does not occur for a non-zero value on top of the data stack
28   ifeq test5_fail
29   goto test5_pass        ; branch did not occur, which is the correct behaviour
30
31   test5_fail:
32   bipush 5               ; push an identifier on top of data stack indicating  that this test has failed
33   breakpoint
34
35   test5_pass:
36   bipush 99              ; 99 signifies that the test has passed
37   breakpoint             ; halt execution and output success to Handel-C debugger window
38
39   .end method
```

Figure 15 Unit test for ifeq bytecode.

Firstly, the test sets up the SB and SP internal registers so that method arguments and local variables can be accessed during the test via *load* and *store* JBCs. However, there are no method arguments or parameters for this test due to its simplicity. Therefore both SB and SP are initialised to 0. This can be seen on lines 12 – 16 in Figure 15.

[14] It can be observed that the *ifeq* test in Figure 15 makes use of other JBCs, for example, *bipush* and *iconst_0*. This means that some unit tests were dependent on others. A number of very basic tests had to be developed to begin with; hence the unit testing was hierarchical in nature.

Next, the *ifeq* JBC is tested to ensure that it branches to the specified target if the top data stack word is equal to zero, and that it does not branch if the top data stack word is non-zero. The code for this can be seen on lines 21 – 29 of Figure 15. Once the JBC unit tests produced positive results, a simple Java program was used to further validate the JVM.

5.3 Simple Example

The simple program shown in Figure 16 (repeated from Figure 7 in Section 4.3) was used to demonstrate the operation of the JakHarta platform in general and the new instructions in particular.

```
1    public class Caller
2    {
3        public static int result;
4
5        public static void main()
6        {
7            demo();
8        }
9
10       public static void demo()
11       {
12           int local1 = 1;
13           int local2 = 2;
14
15           Callee.calc(local1, local2, 3);
16       }
17   }
18
19   public class Callee
20   {
21
22       public static void calc(int arg1, int arg2, int arg3)
23       {
24           int local1 = arg1;
25           int local2 = arg2;
26           int local3 = arg3;
27
28           Caller.result += (local1 * local2 + local3);
29       }
30   }
```

Figure 16 Procedural Java program using static methods and variables.

It can be seen that the Java program in Figure 16 is made up of more than one class and each class has a number of static methods and variables. Static methods can contain local variables, which are automatically allocated and de-allocated on the data stack

when the function is entered and returned from. On line 3, the static class variable *result* is defined, which belongs to the class *Caller*. The static class variable is then manipulated on line 28. This means that the *getstatic* and *putstatic* bytecodes are exercised. Lines 7 and 15 invoke static methods. This results in the *invokestatic* and *return* bytescodes being used. Lines 12-13 and 24-26 use local variables and method parameters which means that *iload* and *istore* bytecodes are also used.

The program of Figure 16 was compiled and romized using the tool chain discussed in Section 4.5. The program was then executed via a Handel-C simulation of JakHarta. Figure 17 shows how the runtime data structures (RTDS) representing the program in Figure 16 were laid out in memory by the romizer and boot loader. This is in accordance with the JakHarta memory map described in Section 4.6, Figure 12.

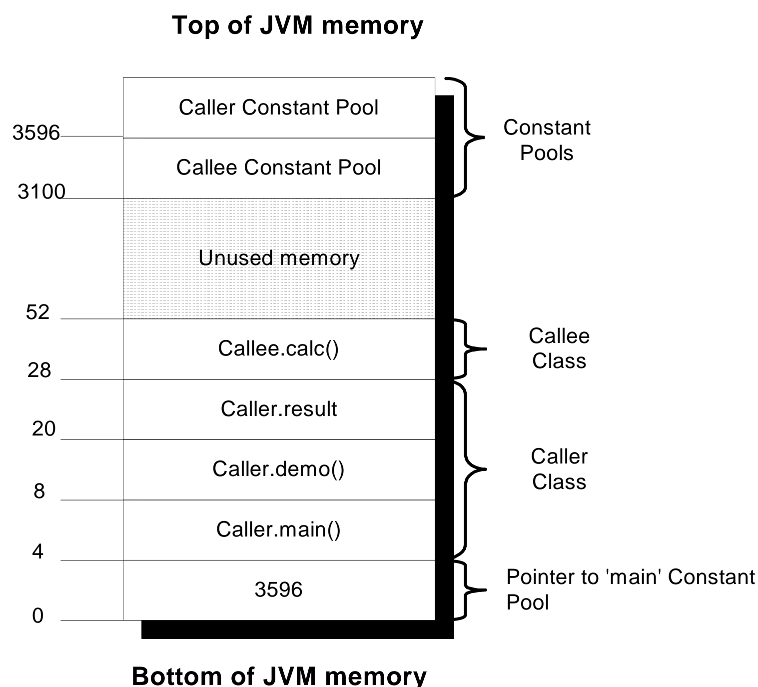

Figure 17 Memory layout of simple Java program.

Results in terms of screen shots were obtained during execution. These were taken prior to and after the execution of a new instruction in order that the state of the JVM could

be noted and the number of clock cycles taken for the new instructions could be calculated.

Figure 18 is the first screen shot which shows the state of the data and return stacks, the internal registers and the clock cycle count prior to calling *demo* on line 7 of Figure 16.

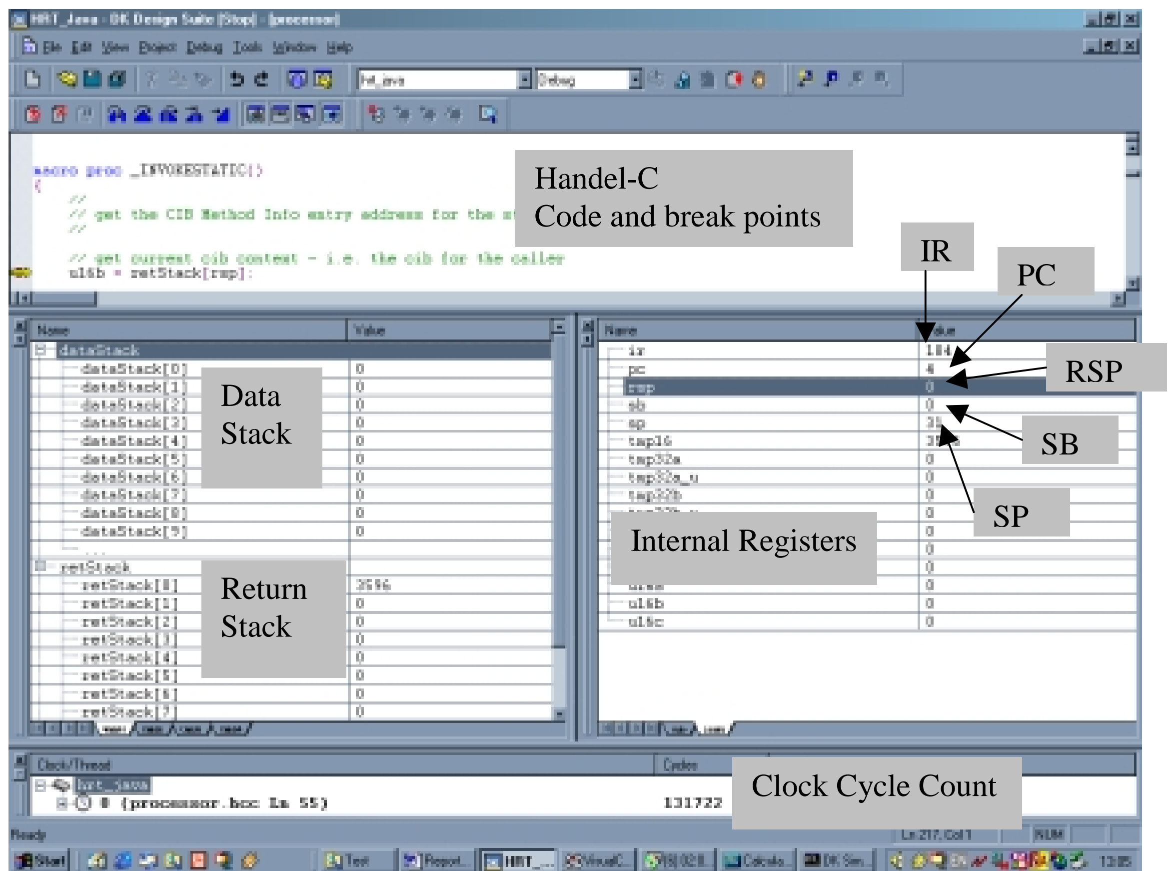

Figure 18 *Main* method before calling *demo*.

The program counter (PC) has a value of 4. As can be seen in Figure 17, this is the memory location where the romizer locates the bytecode for the *main* method and it is the address that JakHarta jumps to as the final part of the boot-up procedure (see Section 4.6). The data stack is empty and the stack pointer (SP) register is set to its initialisation value of 31[15]. The stack base pointer (SB) is initialised to 0 and currently

[15] The data stack pointer (SP) register is initialised to 31 due to the fact that bytecodes that push values onto the data stack pre-increment SP before writing to the data stack. Therefore when the first item is pushed onto the data stack, SP is pre-incremented which causes the value to roll over to 0, which is the intended first stack element.

unused as no static methods have yet been called. The return stack pointer (RSP) is set to 0. The content of the return stack referred to by RSP is the current constant pool address for the *Caller* class. This has been placed at memory address 3596 by the romizer and boot loader and can be seen in Figure 17. The instruction register (IR) contains the value of the *invokestatic* bytecode, which is the first bytecode to be executed. The clock cycle counter is set to 131722, which is the number of clock cycles taken to arrive at the current point of execution[16]. Figure 18 shows that JakHarta boots up correctly and passes control to the *main* method. Figure 19 is the screen shot taken just after the invocation of *demo* on line 7 of Figure 16.

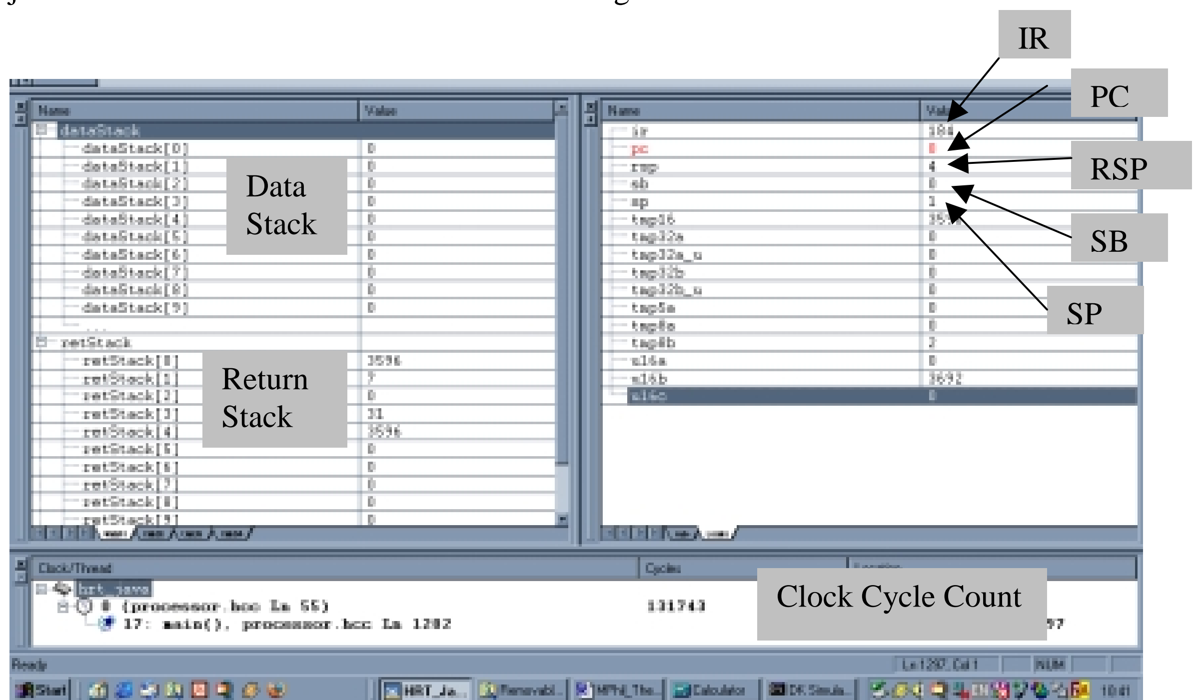

Figure 19 After calling *demo*.

It can be seen that the PC in Figure 18 was 4 and after the *invokestatic* in Figure 19, it is 8. This is the start address in memory of the bytecode for the *demo* method (as can be seen in Figure 17) and it means that JakHarta will start to execute it. It can be seen from Figure 17 that the length of the *main* method is 4 bytes in total (the *invokestatic* JBC is 3 bytes in length and an implicit *return* that is automatically *inserted* by the Java

[16] The clock cycle count at the end of the boot-up procedure is 131722. The reason for this being so high is because RAM and the data and return stacks are initialised to 0. Other initialisation such as loading the program in Figure 16 into RAM also contributes to this value.

compiler is 1 byte in length). Since the *demo* method is the *second* method in the Caller class, it is located directly after the *main* method. Hence its start address is 8. SB is set to 0, which is used to index the data stack element used to store the first local variable *local1*. SP is set to 1 and is used to index the data stack element where the last local variable in *demo* is stored. These values are correct since the *demo* method has 2 local variables. The return information for *main* has been saved on the return stack at elements 1 to 3. Element 1 contains the return address which will be written to the PC when the *return* bytecode is executed. The value of this is 7, which is the value of PC when the *invokestatic* bytecode was decoded, which was 4, plus the length of the *invokestatic* instruction, which is 3 bytes. Elements 2 and 3 of the return stack contain the preserved SB and SP values so that they can be restored upon return from *demo*. The constant pool address for the *Caller* class is at element 4 of the return stack and as *demo* is contained within the same class as *main*, this is the same as the constant pool address at element 0. The clock cycle counter in Figure 19 is set to 131743. The difference between this and the clock cycle count in Figure 18 is 21. This is the number of clock cycles taken to execute the *invokestatic*. This means that the *invokestatic* bytecode behaves in accordance with the design discussed in Section 4.7 and Tables 3 and 4. It also confirms that the number of clock cycles taken to perform the *invokestatic* was as expected.

From the Java program in Figure 16 line 15, it can be seen that the next step is for the *demo* method to call the *calc* method in the *Callee* class. The screen shot in Figure 20 shows the state of JakHarta just before the execution of the *invokestatic* bytecode.

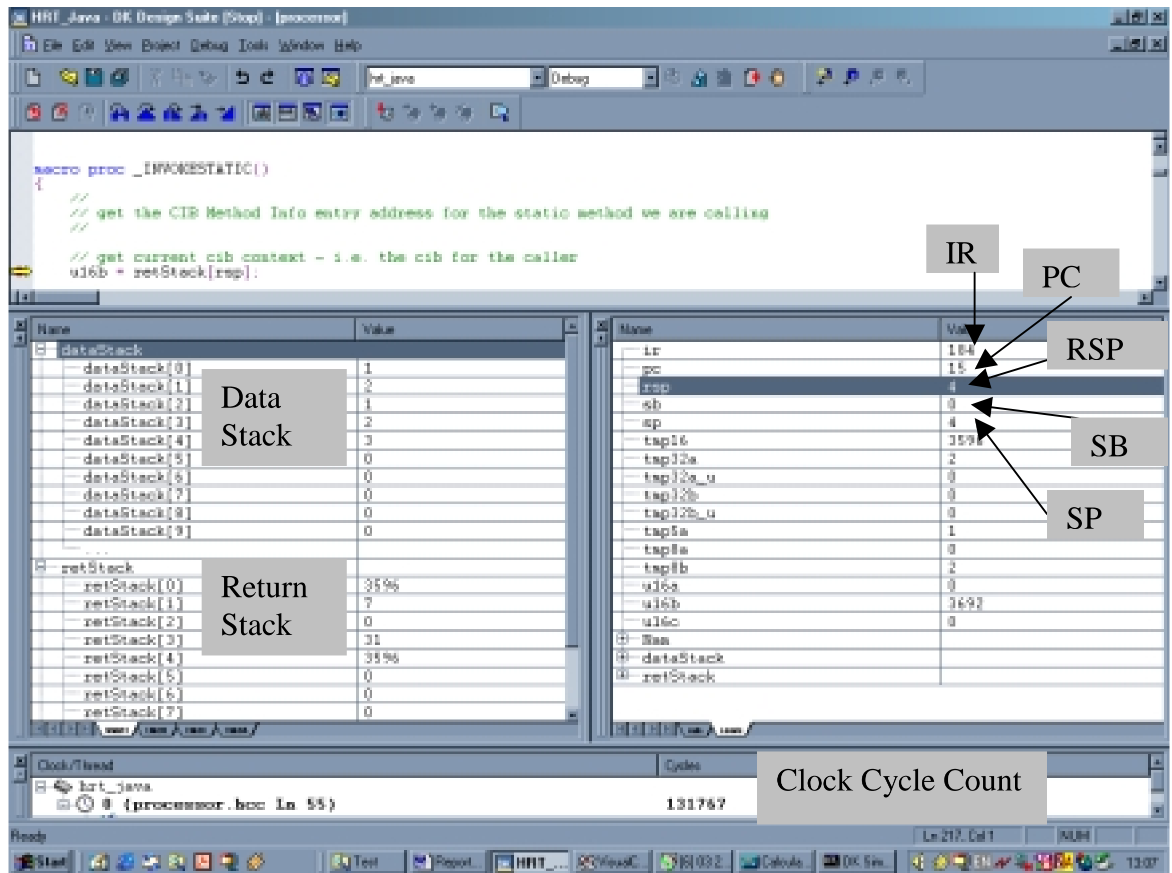

Figure 20 Before calling the *calc* method.

The PC is set to 15 which (as seen in Figure 20) is the memory location for the *invokestatic* for *calc*. The IR contains the value for the *invokestatic* bytecode. Elements 0 and 1 of the data stack store the local variables for the demo method, as can be seen on lines 12 and 13 of Figure 16. Elements 2 – 4 of the data stack contain the arguments passed to calc[17]. As in Figure 19, SB still refers to the first local variable for demo. However SP now refers to the last method argument pushed onto the stack ready for the *invokestatic*. It can be seen that the return stack and RSP remain the same as in Figure 19 as the *invokestatic* has not been executed yet. Figure 20 illustrates that the correct method arguments have been pushed to the data stack prior to the operation.

---

[17] As noted in Section 4, if a method has *N* arguments, then the Java compiler plants code to push them on to the data stack prior to the invokestatic bytecode. Arguments and local variables are then accessed via the iload and istore bytecodes.

In order to determine that the *invokestatic* for *calc* was performed correctly the following screen shot was taken just after the *invokestatic* operation for *calc*.

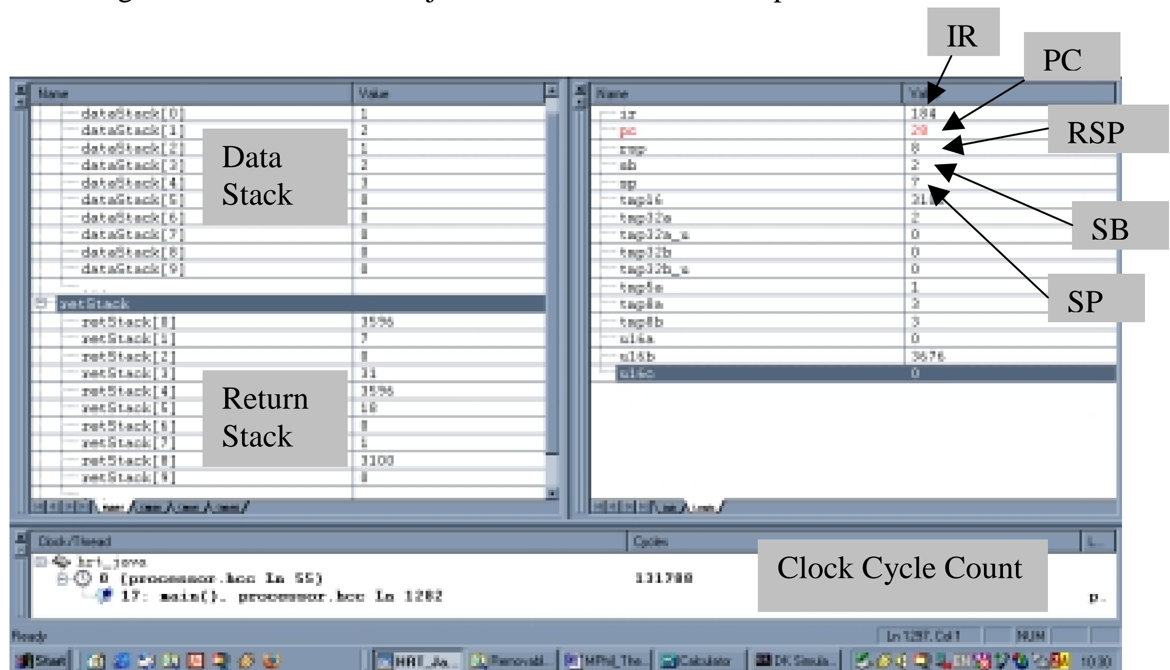

Figure 21 After the *invokestatic* for *calc*.

From Figure 21, it can be seen that the PC is now set to 28. As Figure 17 illustrates, this is the start address of the method bytecode for *calc*. It can be seen on lines 22 and lines 24 – 26 of Figure 16, that the *calc* method has 3 method arguments and 3 local variables. SB is set to 2 (slots 0 and 1 of the data stack are used to store the 2 local variables for *demo*) and indexes the first method argument on the data stack: the value of which is correctly set to 1. SP is set to 7 and indexes the last local variable for *calc* on the data stack. RSP is set to 8 since a new method activation record of 4 stack words in length has been created. RSP indexes the return stack element that contains the CP address for the *Callee* class (the owner of *calc*). As illustrated in Figure 17, this is correctly set to memory location 3100. It can also be noted that the return information for the demo method has also been saved to the return stack. Element 5 of the return stack contains the return address for the *demo* method. Element 6 contains the SB value and element 7 contains the SP value for the *demo* method.

The clock cycle counter in Figure 21 is set to 131788. The difference between this and the clock cycle count of 131767 in Figure 20 is 21. This is the number of clock cycles taken to execute the *invokestatic* for *calc*. This shows that the number of clock cycles taken to invoke *calc* is the same as the invocation for *demo,* even though *calc* has 3 method arguments and 3 local variables whereas *demo* has none of either. The *invokestatic* executes in constant cycle time regardless of the number of method parameters or local variables, and is therefore fully deterministic as designed.

The following screen shot shows the *calc* method just before the *getstatic*. This corresponds to line 28 of Figure 16 where the value of the static class variable *Caller.result* is read.

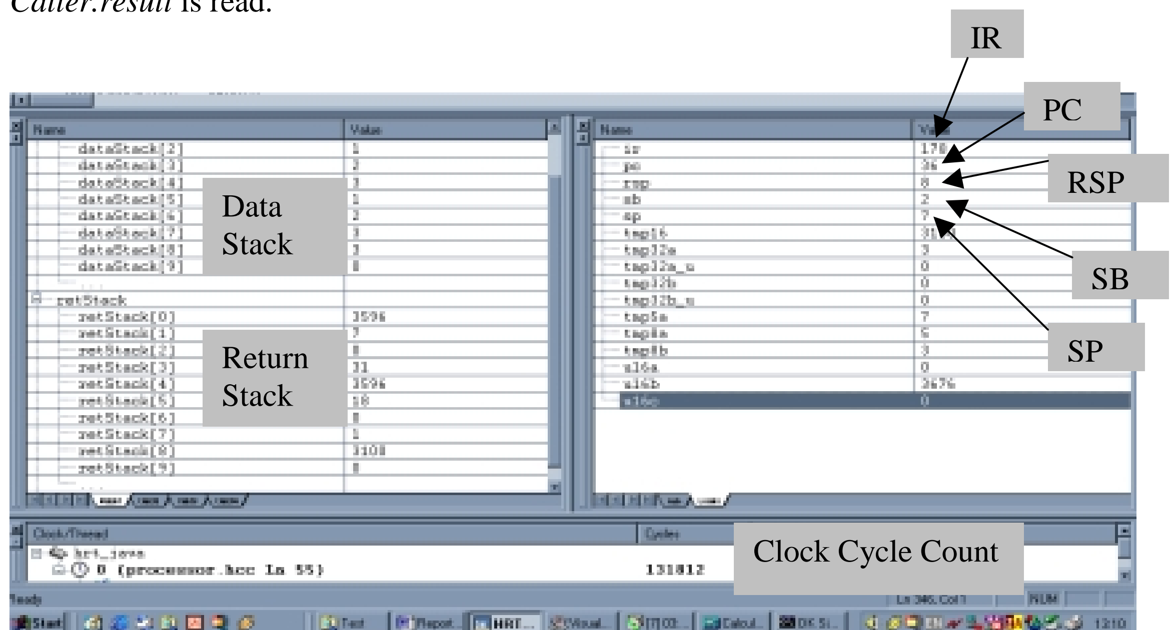

Figure 22 *calc* method prior to the *getstatic*.

The IR is set to 178, which is the value of the *getstatic* bytecode. Recall from Figure 17 that the start address for the JBC for calc is 28. The offset of the *getstatic* JBC in the *calc* method is 8. Therefore the PC is set to 36, which is the correct memory address of the instruction. Data stack elements 2 – 4 contain the method arguments passed to the current method, which is *calc*. Data stack elements 5 – 7 contain the local variables, *local1*, *local2* and *local3,* which have now been initialised to the values 1, 2 and 3 respectively. The initialisation code for this can be seen on lines 24 – 26 of Figure 16.

SB is set to 2 and references the first parameter passed to the method, which is *arg1*. SP is set to 7, which references the last local variable for the *calc* method, which is *local3*.

In order to ensure the correct implementation of *getstatic* and to determine the number of clock cycles taken, another screen shot was taken immediately after its execution.

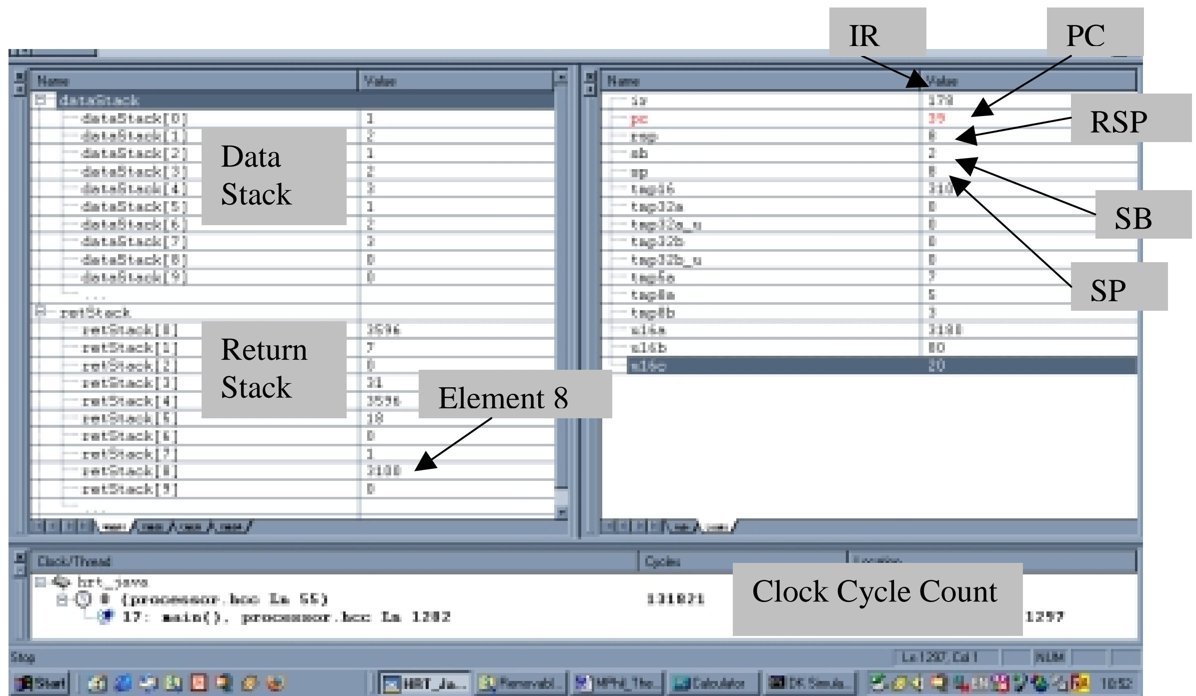

Figure 23 After *getstatic* in *calc*.

It should be noted that prior to the execution of *getstatic*, the value of SP was 7. From Figure 23, after the execution of *getstatic* the value is 8. It can also be seen that element 8 of the data stack contains 0, which is the current value of the static class variable *result* which belongs to the *Caller* class. Prior to executing the *getstatic* bytecode, the clock cycle count was 131812 as can be seen in Figure 22. After executing the *getstatic* bytecode, the clock cycle count was 131821 as can be seen in Figure 23. Therefore the implementation of *getstatic* executes in a total of 9 clock cycles.

As discussed above, the implementation of *getstatic* behaves as expected. However, line 28 of Figure 16 also results in the generation of a *putstatic* JBC. Figure 24 is a screen shot taken prior to the execution of the *putstatic*.

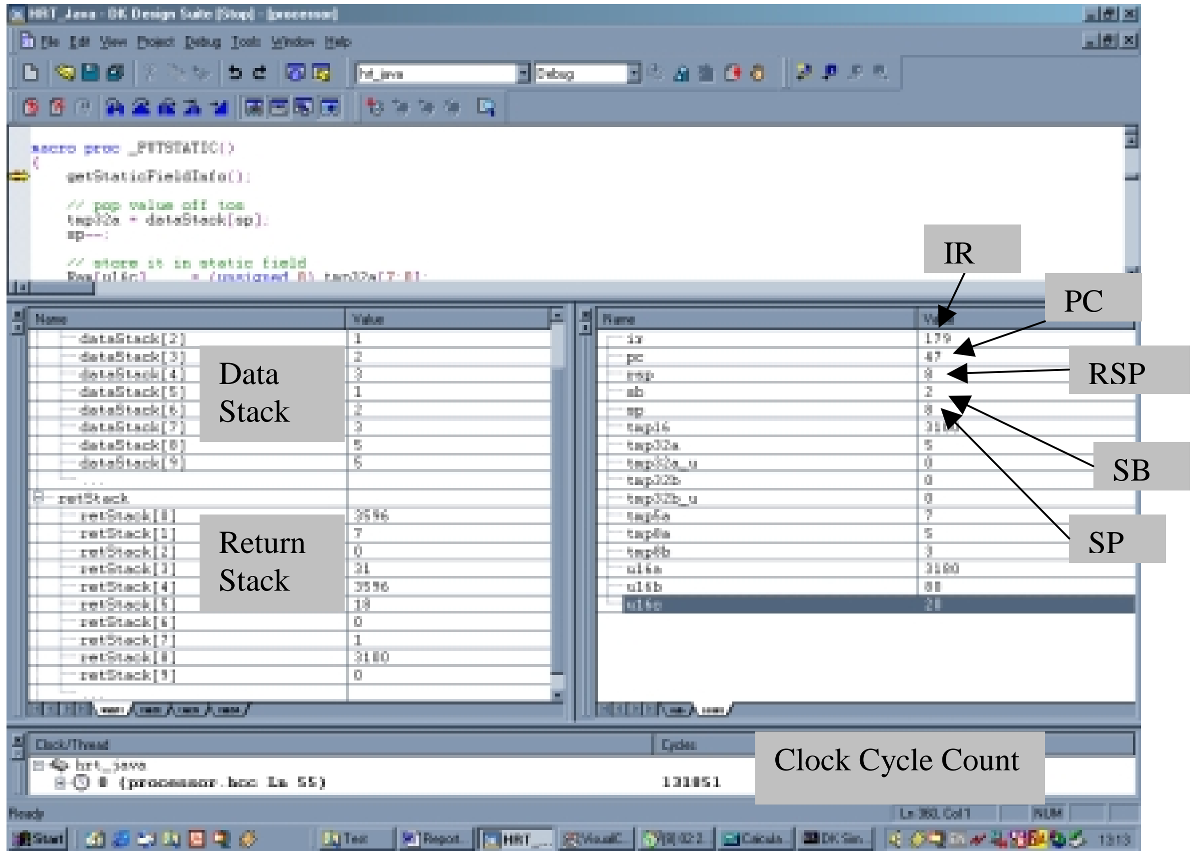

Figure 24 ***calc*** **method prior to** ***putstatic*****.**

The IR is set to 179, which is the value of the *putstatic* bytecode. Figure 17 shows that the start address for the JBC for calc is 28. The offset of the *putstatic* JBC in the *calc* method is 19. Therefore the PC is set to 47, which is the correct memory address of the instruction. Of greater interest is the value of SP, which is set to 8. This references the value 5 on the data stack, which is to be written to the static class variable *Caller.result*. This is described in Section 4.7.3. Note that this is a temporary value used for the assignment and is discarded as part of the behaviour of the *putstatic* bytecode by decrementing SP. In order to verify the correct execution of the *putstatic* bytecode, a breakpoint was set just after its execution. The screen shot in Figure 25 was then taken.

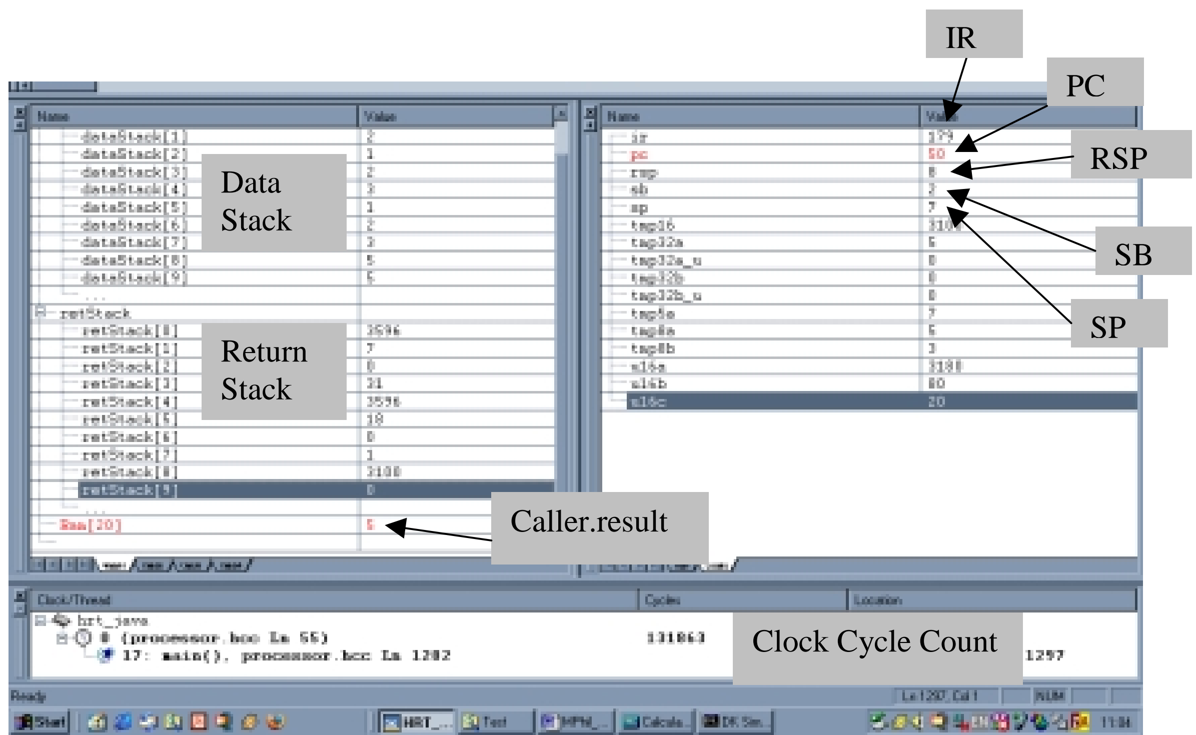

Figure 25 After the *putstatic* in *calc*.

It can be seen that in Figure 24, SP was set to 8 and that after the execution of the *putstatic* in Figure 25, it has been decremented and has the value 7. This means that the temporary value 5 has been discarded. It was confirmed that the value 5 was written to the correct memory location representing the static field *Caller.result*. This was memory address 20 and can be seen in Figure 17 and Figure 25.

Prior to executing the *putstatic* bytecode, the clock cycle count was 131851 as can be seen in Figure 24. After executing the *putstatic* bytecode, the clock cycle count was 131863 as can be seen in Figure 25. Therefore the *putstatic* bytecode executes in 12 clock cycles as expected.

After the calc method finishes executing, a *return* instruction is encountered in the bytecode stream. This corresponds to the implicit *return* on line 29 of Figure 16 that is automatically inserted by the Java compiler. The *return* instruction will transfer program flow back to the caller which is the *demo* method owned by the *Caller* class. In order to validate the correct implementation of the *return* bytecode a screen shot was captured prior to its execution. This can be seen in Figure 26.

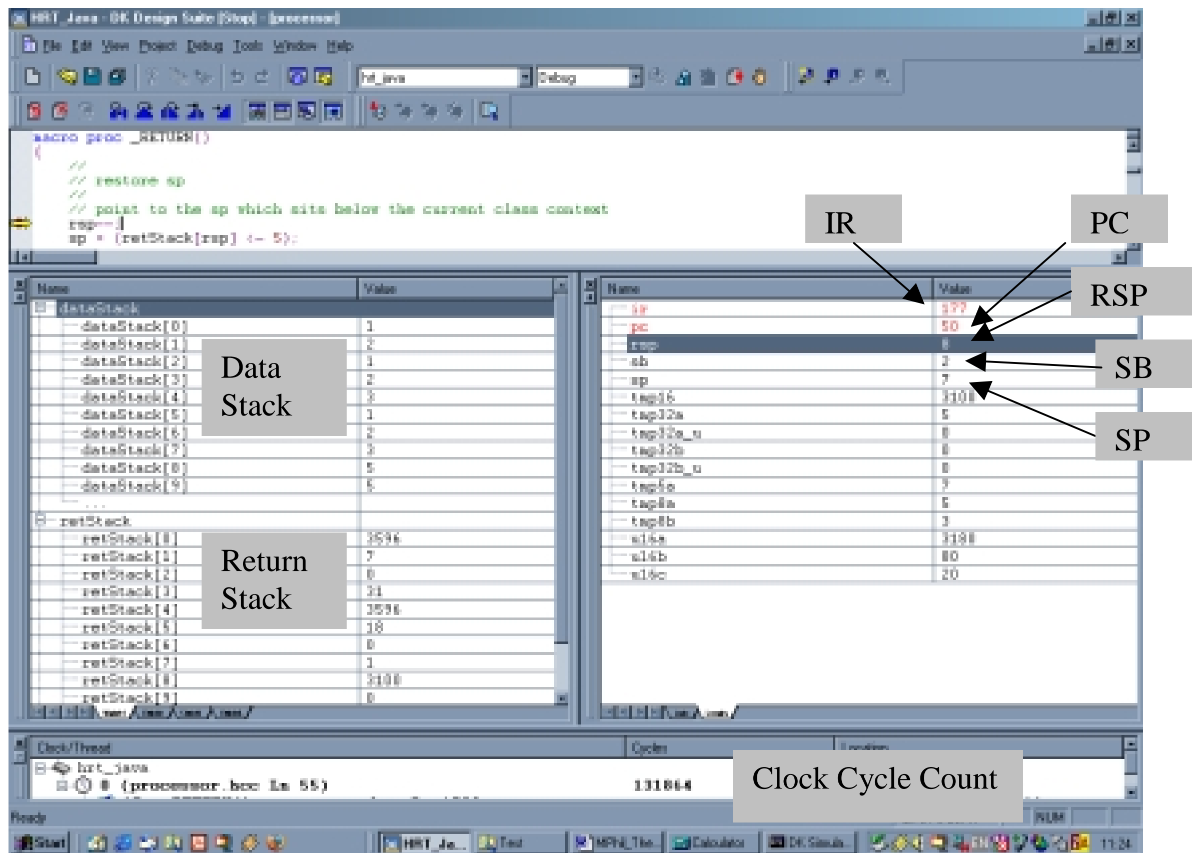

Figure 26 Before the *return* in *calc*.

The IR contains 177 which is the value of the *return* bytecode. The PC contains the location of the *return* in the bytecode stream for *calc*. SB, which references the first method argument of the *calc* method, *arg1*, is set to 2 and SP, which refers to the last local variable of *calc*, *local3*, is set to 7. RSP is currently set to a value of 8 and refers to the constant pool address of the *calc* method, which is 3100. This can also be seen in Figure 17. It should also be noted that elements 5 – 7 of the return stack contain return information, which was saved prior to the execution of *calc*. This includes the saved values of SB, SP and PC that the *return* bytecode must restore in order that the *demo* method can resume execution. The next screen shot was taken just after the *return* in *calc*.

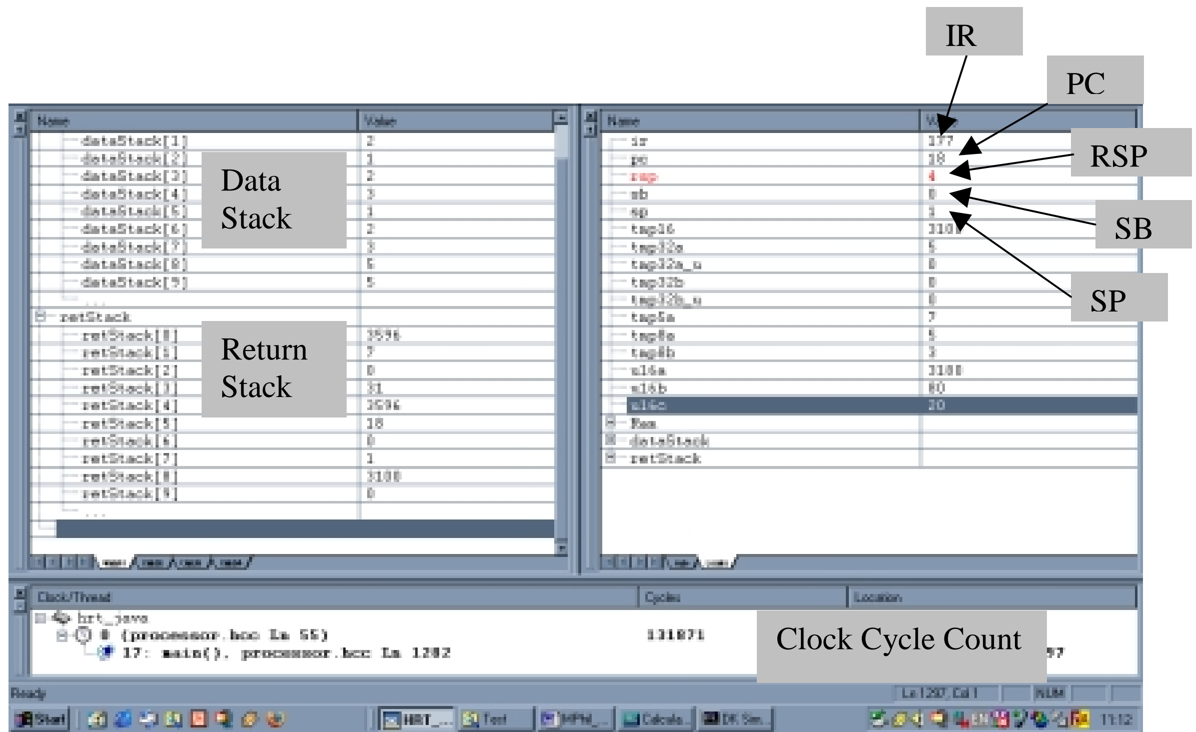

Figure 27 After the *return* in *calc*.

As discussed above, the *return* instruction must restore the execution context of the caller in order that execution can resume. Comparing Figure 26 with Figure 27 after the *return*, it can be seen that the PC, SB and SP have all been restored to the correct values that were saved on the return stack by the *invokestatic*. It can also be noted that RSP has been decremented by 4 and refers to the return stack location that contains the constant pool address of the *Caller* class. The clock cycle count before the *return* was encountered in the bytecode stream is 131864. It is now 131871 which means that the *return* bytecode has taken 7 clock cycles to execute. The implementation of *return* therefore agrees with the design as described in Section 4.7.

By simulating the execution of the simple program in Figure 16 and inspecting the contents of the internal registers and memory during the exercise, it has been demonstrated that the implementation of the new instructions (*invokestatic*, *return*, *getstatic* and *putstatic*) behave as expected and in accordance with the JakHarta design as described in Section 4.7. The simulation also shows that each new instruction executes in the expected number of constant clock cycles and is therefore fully deterministic.

5.4 Complex Example

A second more complex example program was developed. This was done for a number of reasons, first being to confirm the results obtained for the simple example. The second reason was to provide a more challenging test case. Hence the second example is recursive, since recursion stresses the method invocation mechanism by creating a number of repeated data and return stack method activation records and then restoring them at the appropriate points as the recursion *unwinds*[18]. Any errors in the *invokestatic* and *return* bytecodes are more likely to be manifested in the complex example, since this would affect program flow. Similarly any stack elements that are incorrectly allocated or used are likely to be more noticeable since a thorough check of stack usage can be made at various points during the recursion. The third reason for using a recursive example was to demonstrate that the WCET can be precisely calculated for method invocations. As discussed in Section 1.5.1, the ability to calculate the WCET is an important requirement for a HRTS. This discussion will focus on the calculation and verification of the WCET for the complex example.

Figure 28 illustrates the complex example that uses recursion in order to calculate the factorial of the given number. The factorial function, denoted as $n!$, describes the operation of multiplying a number by all the positive integers smaller than it. For example, 5! = 5 * 4 * 3 * 2 * 1. The terminating case which stops the recursion from continuing indefinitely can be seen on line 23, where the value of n is equal to 1. The general case is when $n > 1$, meaning that the equation and recursive definition $n! = n * (n - 1)!$ is performed. This can be seen on line 19. Each recursive call moves the computation one step closer to termination.

[18] In Section 1.5.1 it was highlighted that the use of recursion is generally non-deterministic since the terminating condition is usually dependent on input data. Hence the depth of recursion, and therefore tight stack usage and execution time cannot be known at compile time. Programming techniques such as those discussed in [PUS02B] are often used when developing HRTSs in order to prevent input data dependencies. However a recursive programming example is a valuable test since it stresses the implementation of the method call and return mechanism.

```
1       public class FactTest
2       {
3              public static int result;
4
5             public static void main ()
6             {
7                    FactTest.calcFactorial ();
8             }
9
10            public static void calcFactorial ()
11            {
12                   FactTest.result = FactTest.factorial (6);
13            }
14
15            public static int factorial (int n)
16            {
17                   if (n > 1)
18                   {
19                           return FactTest.factorial (n - 1) * n;
20                   }
21                   else
22                   {
23                          return 1;
24                   }
25            }
26       }
```

Figure 28 Recursive program which calculates the factorial of a given integer.

The WCET for the *factorial* method can be calculated with the aid of a directed acyclic graph (DAG). The DAG separates the JBCs for the factorial method into a number of basic blocks. Figure 29 shows the DAG, which represents the disassembled bytecode for lines 15 to 25 in Figure 28.

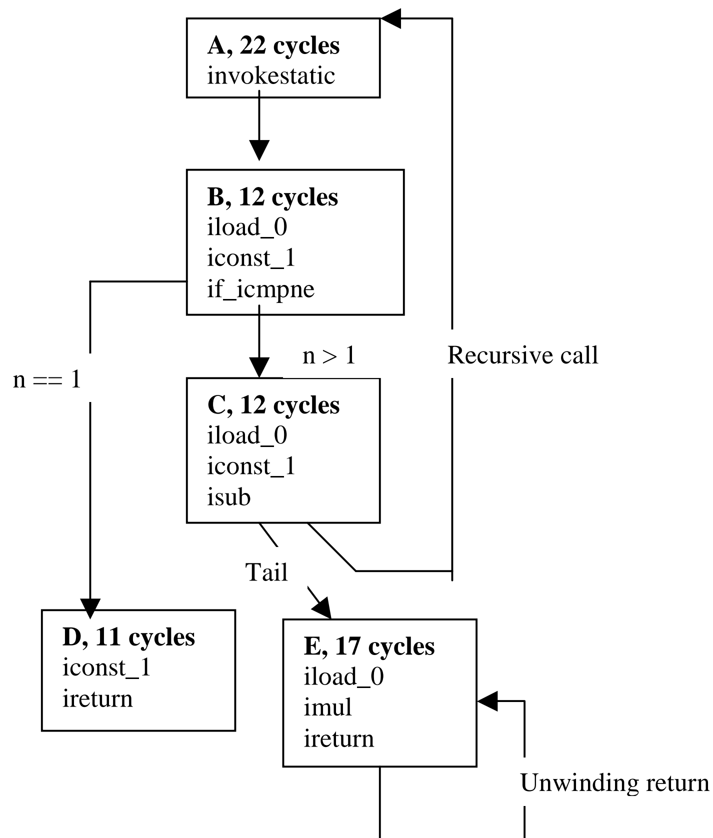

Figure 29 DAG for recursive factorial program.

Each block labelled A, B, C, D and E specifies the number of clock cycles used for execution of the block. This includes an additional 1 clock cycle per instruction for fetching of the next instruction from memory. The clock cycles were obtained from the instruction clock cycle table in Appendix 1.

Block A represents the invocation of the factorial method as seen on lines 12 and 19 in the Figure 28. This *factorial* method is called first on line 12 and then recursively on line 19. Hence Block A represents the method call overhead. Block B represents line 17 where n is checked to see if it is greater than 1. If $n > 1$ then block C is executed, otherwise block D is executed.

Block C represents part of line 19, where 1 is subtracted from *n*, ready for the recursive call to Block A. Block D represents line 23 where 1 is returned if n is equal to 1. Block E is the tail end of the recursion (multiplication by *n*) and is executed as the stack

unwinds. This means that the recursive calls return, and the method activation records are restored and popped off the data and return stacks.

Expression 1 can be used for calculating the number of clock cycles depending on the initial value of the method parameter *n*.

$$((A + B) * n) + (C * (n - 1)) + D + (E * (n - 1))$$

Expression 1 WCET Calculation for factorial method.

Expression 1 can be simplified. This is shown in Expression 2.

$$((A + B) * n) + ((C + E) * (n - 1)) + D$$

Expression 2 Simplified WCET Calculation for factorial method.

When n is 2, this will require only 1 recursive call and the number of clock cycles taken can be precisely calculated as:

((22 + 12) * 2) + ((12 + 17) * 1) + 11 =

68 + 29 + 11 = 108.

To validate the calculation, before and after screen shots of the Handel-C debugger were taken in order to record the clock cycle count. Figure 30 is the screen shot taken prior to first *invokestatic* for *factorial*.

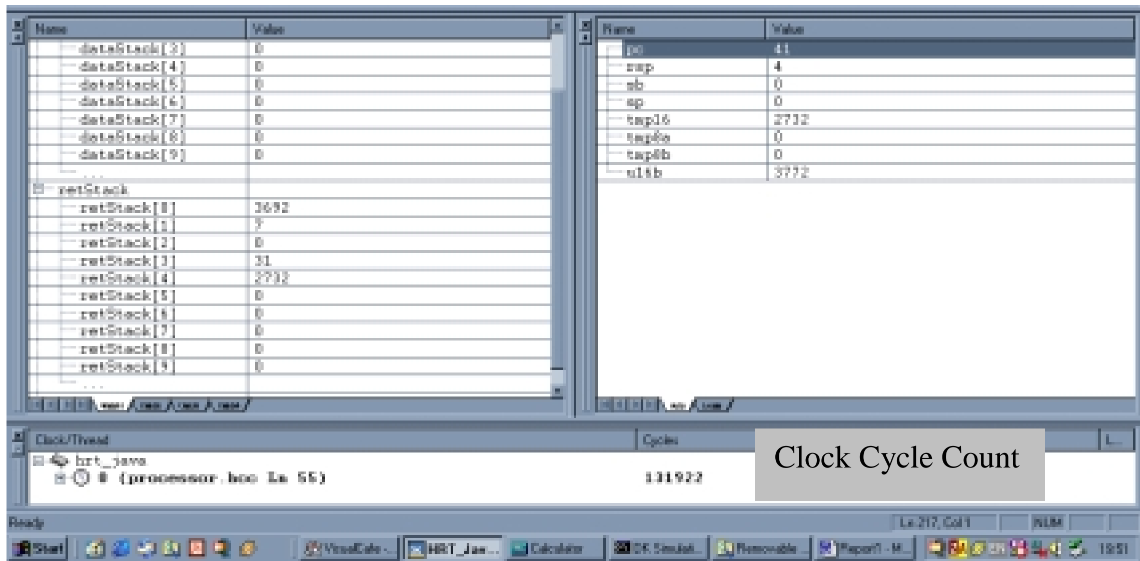

Figure 30 Prior to the first invokestatic for *factorial*.

It can be seen that the clock cycle count is 131922. Figure 31, shows the screen shot taken after the *final return* for the *factorial* method.

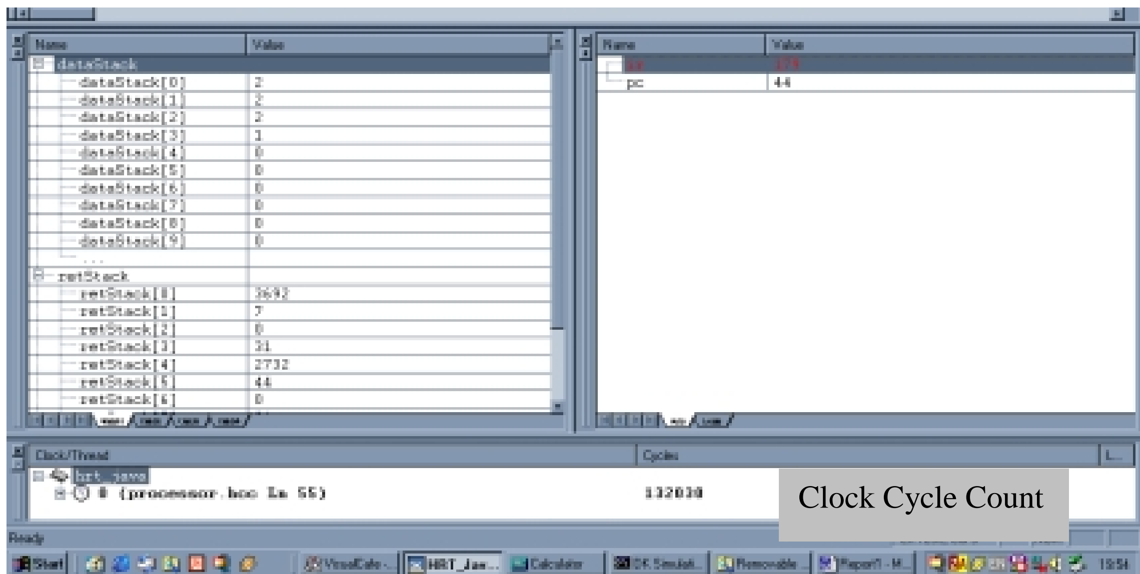

Figure 31 After the final return from *factorial*.

The difference in clock cycles between Figure 30 and Figure 31 is 108. This is agrees the calculated number of clock cycles.

The program in Figure 28 was executed for values of n between 1 and 6. In all cases the number of clock cycles taken agreed with the calculated number of clock cycles using the Equation 2. When n is 6, 5 recursive calls are required. Due to the current limitation of 32 words for both the data and return stacks, any higher values of n would result in a stack overflow error. However, it should be noted that because JakHarta has been implemented as a reconfigurable processor in FPGA the stacks can be set to the worst-case sizes for a HRTS. This simply requires a recompile of the Handel-C source code.

5.5 Summary

Unit testing was used to demonstrate the correct behaviour of individual JBC instructions. These were programmed in Java assembly language using a modified version of the Jasmin assembler, which meant that internal JakHarta registers could be accessed and the JBC generated could be explicitly controlled.

A simple Java program showed that the existing JBCs and the new JBCs worked correctly in a proper context. It also showed that the number of clock cycles taken to execute the new JBCs was constant time and agreed with the design in Section 4.7.

A more complex Java program was then used as a challenging test case. This stressed the *invokestatic* and *return* JBCs by ensuring that the creation and destruction of method activation records on the data and return stacks worked correctly during recursive calls and returns. This also demonstrated that the number of clock cycles taken during recursive method invocations can be calculated precisely provided that the input data dependencies are known.

A WCET analysis tool could be developed in order to perform these calculations. This would parse the JBC for a method and employ a look-up table to determine the execution time of each JBC. These could then be added to determine the total execution time for the method.

6 Conclusions and Future Work

6.1 Thesis Conclusions

HRTSs have strict behavioural and timing requirements, which must be met under all runtime situations. Many HRTSs are also safety critical and as such, a practical Java solution must also be suitable for safety critical systems. HRT practices include static analysis in order to ensure *a priori* that a system will meet its requirements. This is a fundamental difference when compared with SRTSs. Because the consequence of a missed deadline for a SRTS is less severe, they usually employ somewhat simpler and cheaper ad-hoc empirical techniques instead. This means there are a number of consequences for HRTSs. When compared with SRT development, HRTSs take much longer to develop, require a much higher skill level and are much more expensive. In order to enable static analysis of programs, HRTSs must carefully control complexity and minimise non-determinism at both the software and hardware levels. Developing HRTSs requires meticulous attention to detail and a different mind-set from that needed when developing general-purpose and SRTSs.

Java has a number of advantages for HRTSs development. The Java language is easy to learn, well defined and very popular in other areas of software development. However, standard Java conflicts with the needs of HRTSs in a number of ways. Key features that have contributed to the popularity of Java are unnecessary and unsuitable for HRTSs. These include garbage collection, dynamic class loading and the Java class libraries. Java also lacks certain key features that are essential for HRT development, for example, the ability to expose the underlying hardware and program devices at a low level.

A number of attempts have been made at making Java suitable for HRT. The Real Time Specification for Java has addressed a number of shortcomings in standard Java for RT development. However, it appears to have done this by augmenting standard Java with additional features. The RTSJ remains backwards compatible with standard Java in that a program written for standard Java will also execute in a RTSJ environment. It appears

from [DIB02] that the RTSJ group may have been restricted in the changes that they could make. One of the original designers of Java, James Gosling, pointed out at an early meeting, "if we [the group] crossed some undefined line in the changes, it would not matter what we called it. It would not be Java". Whilst the RTSJ may be appropriate for some SRTSs it is not suitable for HRTSs.

In order to overcome the problems with the RTSJ with respect to its usage in HRTSs, the Puschner profile [PUS02A] has removed a number of the undesirable features and simplified the execution model. Whilst the profile is a positive improvement on the RTSJ, it is the author's view that further constraints are necessary. For example, the Puschner profile retains support for dynamic class loading. As discussed in Section 2.4, this is unnecessary and unsuitable for HRTSs and should be removed. The Puschner profile uses a non-real-time initialisation phase to contain features that are problematic during static analysis. Dynamic class loading is one of the features executed in the initialisation phase. For reasons discussed in Section 2.7, it is argued that all phases of execution must be made deterministic for a HRTS. Given that dynamic class loading and bytecode verification are not required, it is suggested that the initialisation phase can be made fully deterministic**.** This is the intended approach for JakHarta.

The implementation aspects and consequences at the JVM level should be carefully considered. By understanding the JVM at this level it is possible to identify sources of non-determinism and features that may cause difficulty during static analysis and certification.

The practical work outlined in this thesis contributes to the development of a JVM suitable for HRT. The JVM, named JakHarta, has been developed in Handel-C for deployment on an FPGA. A tool chain has been developed in order that simple HRT applications and JVM test programs can be compiled, assembled, loaded and executed. This was not possible with the original Oxford JVM. Each individual instruction was tested to create a reliable basis for the development of JakHarta. In order to support procedural programming comparable with C, encapsulation and multiple source file development, the *getstatic*, *putstatic*, *invokestatic*, *return* and *ireturn* JBCs were

implemented in Handel-C for execution in FPGA hardware. These utilise the deterministic runtime data structures generated by the romizer so that they execute in constant cycle time. Simple HRT Java applications can be supported provided they are programmed using static techniques outlined in [PUS02B]. This was not realistically possible with the original Oxford JVM.

6.2 Future work

In order to develop JakHarta into a full JVM for HRT, the remaining additional work presented in Section 3.3.2 needs to be completed. However, there is still other work that should be considered in order to make JakHarta a complete solution. JakHarta does not meet any specific HRT Java standard. However, as discussed in the thesis, those that exist or which have been proposed are considered unsuitable. At the time of writing this thesis, the Safety Critical Java standard is being developed by the Open Group [OPE01]. It would be beneficial to review the Safety Critical Java standard once complete and, if appropriate, modify JakHarta in accordance with it. It should also be worth considering providing hardware support for debugging and for supporting languages other than Java.

Section 4.6 discussed the Java constant pool in the class file and the equivalent runtime constant pool implemented by JakHarta. The main advantage of the runtime constant pool is that the JBC for a method does not need to be parsed so that constant pool indexes can be replaced by memory addresses. However, the runtime constant pool requires an extra level of indirection when executing complex instructions such as *invokestatic*. An alternative approach could be taken for JakHarta that removes the need for the runtime constant pool; hence the constant pool runtime data structures could be discarded. For example, Section 4.6 describes the design of the runtime constant pool entries for methods. The information required by the *invokestatic* instruction includes the number of method arguments and the number of local variables. This information could be pre-pended to the JBC for a method so that a direct jump can be made to the JBC for the method. The *invokestatic* JBC would have the memory address of the JBC as an argument instead of the constant pool index in order that this could be achieved. A

disadvantage with this approach is that the method bytecode would need to be parsed and modified, which would increase the complexity of the romizer.

As stated above, the current implementation of JakHarta uses Handel-C for deployment on an FPGA. This allowed behavioural modelling of JakHarta with minimum learning time due to the simplicity of the Handel-C language. However, the issues surrounding the use of FPGAs and behavioural hardware compilers for HRTSs are unknown to the author. Many HRTSs would require that the compilers used for development have safety critical certification. This includes compilers used for hardware as well as software, for example the Handel-C compiler. Also the issues in the use of re-configurable cores for safety critical systems are unknown to the author. For example, the data and return stacks for JakHarta can be configured depending on the worst-case stack sizes required for a HRTS. However, this requires a recompilation of the processor source code and could potentially invalidate previous verification and testing results. Since FPGAs are highly configurable and programmed at boot-up, this may be considered too complex for HRTSs, since the more complex the processor is, the harder it is to validate and the higher the likelihood of systematic design errors [ENG97]. For a practical solution, it will be necessary to fully research these issues. It may also be necessary to consider adopting an RTL (register transfer language) implementation of JakHarta using VHDL or another suitable hardware description language (HDL). The safety critical certification costs of a behavioural hardware implementation compared with an RTL approach should also be investigated.

The suitability of OO programming for HRTSs was discussed in Section 2.4. It is clear that fundamental OO features such as polymorphism conflict with the needs of many HRTSs and may be regarded as unnecessary. This suggestion may appear restrictive but is sensible given that the primary considerations for HRTSs are determinism, the ability to apply automated static analysis and, for many, the ability to satisfy safety certification. Features such as polymorphism and overriding may be appropriate for a subset of SRTSs. However they are unnecessary and undesirable for the majority of HRT control applications. This suggestion concurs with studies cited by Booch [BOO94], which states that polymorphism is not needed about 85% of the time. Given

that this statement is made in the context of general-purpose systems, it likely that OO features are even less frequently needed for HRTSs. It may be argued that HRTSs may benefit from OO since they are becoming more complex and OO features help promote better designs, greater re-use and maintainability. However, it is argued by [BIE95] and [GOT02] that the benefits and usage of OO features such as polymorphism are exaggerated and mainly based on anecdotal evidence. It is suggested by [NAS04] that some OO languages have features that could make it difficult or impossible to satisfy the requirements of safety standards such as DO178B. Whilst considering the use of OO for HRTSs development it should also be noted that design concepts which promote quality and maintainability, for example, structured design, modularisation, encapsulation, specification via interfaces, high cohesion and low coupling, are by no means unique to OO and can be achieved using non-OO languages.

Although there may be concerns with the use of OO in HRTSs, the author believes that there may be advantages in providing a HRT JVM that supports a simple object-based model. For example, disallowing overriding in classes would remove dead code associated with this feature. Disallowing overriding would also promote better OO design by restricting inheritance so that it could only be used to extend class behaviour rather than changing it [COA96]. Overriding and multiple inheritance could be disallowed in interfaces, in order to simplify static analysis. Provided that the underlying runtime data structures and JBC instructions are implemented in a deterministic manner, then the option to use polymorphism would still be available for the small subset of systems where it is deemed necessary. Further work is required in identifying the practical situations where polymorphism might be needed in a HRTS and the trade-offs with respect to static analysis and certification. Ultimately, the advantages must significantly outweigh the disadvantages.

In order to support a suitable Java object model a number of issues must be considered. Runtime data structures used to represent objects, arrays and the exception handling mechanism need to be carefully designed for constant or tightly bounded access time. In order that objects and arrays can be created, a suitable runtime heap must be implemented. A fundamental issue with heap design for HRTSs is fragmentation.

Fragmentation can result in an allocation request failing even though the total amount of free memory would be sufficient for the request [SIE02]. Different sized heap blocks allocated contiguously typically cause this. If some of the smaller blocks are de-allocated during program execution, they may not be large enough to satisfy future allocation requests. In desktop operating systems allocated heap blocks can be compacted if an allocation fails because of fragmentation. However this introduces complexity into the runtime system and adds significant non-determinism when creating new objects. However if the approach that all objects are created once during the initialisation phase is taken, this means that the heap can be very simple. Objects are created once during the initialisation phase and are never destroyed, meaning that heap fragmentation is not an issue. Therefore objects can be simply assigned heap space without consideration for fragmentation. The following provides a general discussion of runtime data structure design, for supporting polymorphism and Java interfaces. As discussed in Section 2.4, subtype polymorphism in Java is supported via dynamic binding. This is usually achieved by providing each class with a method table. Figure 32 illustrates a method table design, which allows the *invokevirtual*[19] JBC to execute in a constant number of clock cycles.

[19] Invokevitual is used to call a polymorphic method for a particular object. At a basic level, it uses the reference of the object to gain access to the method dispatch tables (described in Section 2.4). It then calls the appropriate method implementation for the object type.

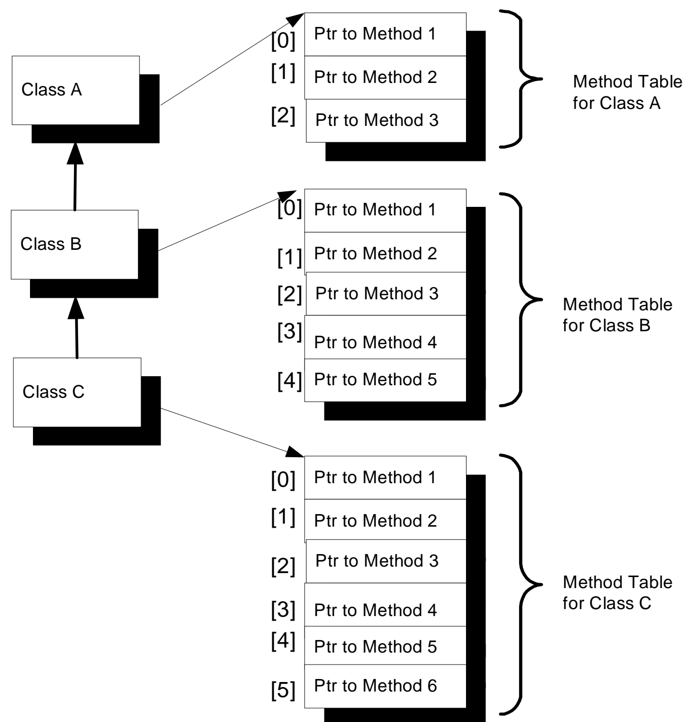

Figure 32 Class diagram showing deterministic implementation of method tables.

It can be seen that super-class method tables are copied down the class hierarchy. Additional methods provided by the sub-class are then appended. These structures enable the *invokevirtual* instruction to execute in constant cycle time since super-class method tables never need to be *searched* and method entries can always be found at the same position within a table. The time taken to perform a method invocation using these runtime data structures is bound constant, regardless of where the method is within the class hierarchy.

It can be observed that the design in Figure 32 is able to support method overriding since sub classes have their own modifiable copy of the super class method table. However, assuming that method overriding is to be disallowed, there can only be one implementation of a method in a class hierarchy. This means that only one method table is actually required for the class hierarchy. This modified design is illustrated in Figure 33.

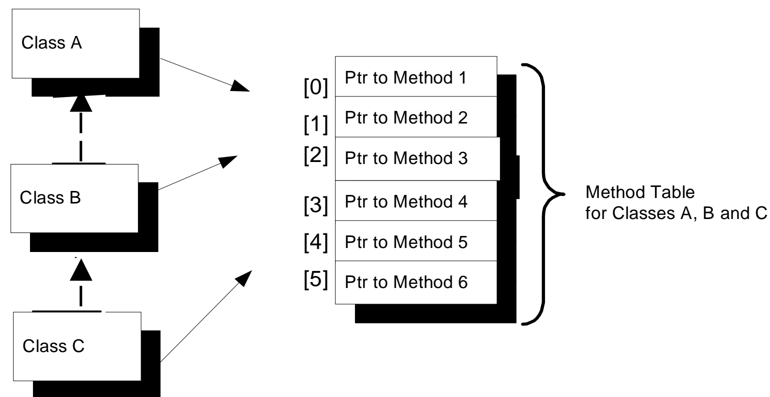

Figure 33 Single method Table for a class hierarchy.

Section 2.3 described the benefits of Java interfaces. However, interfaces are problematic from a HRT perspective, for reasons discussed next. Therefore the design of the runtime data structures representing interfaces, suitable for HRT needs to be considered. A class may implement a number of interfaces and each interface may inherit multiple interfaces. An interface reference can be used in a Java program to manipulate an object, which implements the interface. For example, an interface method can be invoked using the invokeinterface[20] JBC. Given that an interface is essentially an abstract base class, a class implementing a set of interfaces is in effect employing a form of multiple inheritance. Figure 34 shows the method tables for 2 class hierarchies implementing different interfaces.

[20] Invokeinterface is similar to invokevirtual. The Java compiler generates it when an interface method is invoked for an object which is referenced via an interface reference.

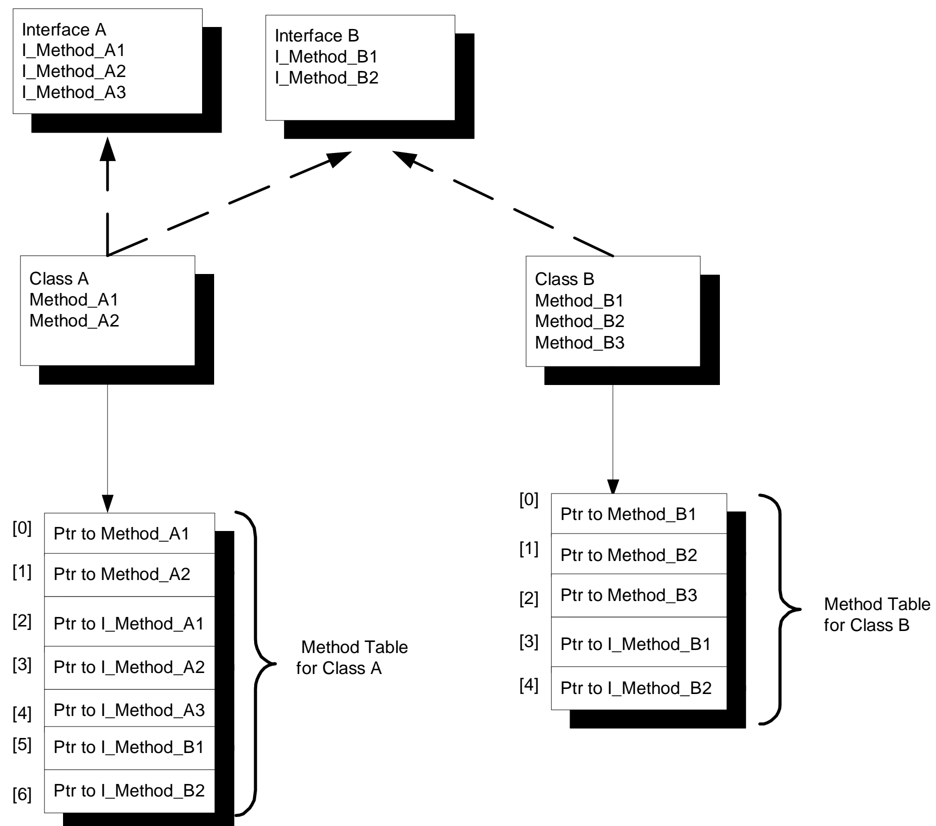

Figure 34 Non-deterministic interface method tables.

It can be seen that Class A *implements* Interface A and Interface B. Class B implements Interface B. Hence, both classes implement Interface B. In this example the method table for Class A is created by adding its own entries first and then appending the method entries for Interface A and then Interface B. The method table for Class B is created by adding its own method entries and then appending the method entries for Interface B. A reference of type Interface B could be used to refer to an object of Class A or Class B and used to invoke a method declared in Interface B using the *invokeinterface* JBC.

It can be seen that methods for Interface B are located in different entries in the method tables for Class A and Class B. This is because Classes A and B implement a different set of interfaces. If such a design was used, it would mean that the method table for a

hierarchy would need to be searched whenever an interface method is invoked, since the location of an interface method entry may differ.

The problems associated with interfaces mean that JVM implementations usually employ separate method tables for interfaces. These can be called *interface method tables*. Therefore each class would have an interface method table (as well as a normal method table), provided that the class implements at least one interface. Figure 35 illustrates a deterministic design for interface method tables.

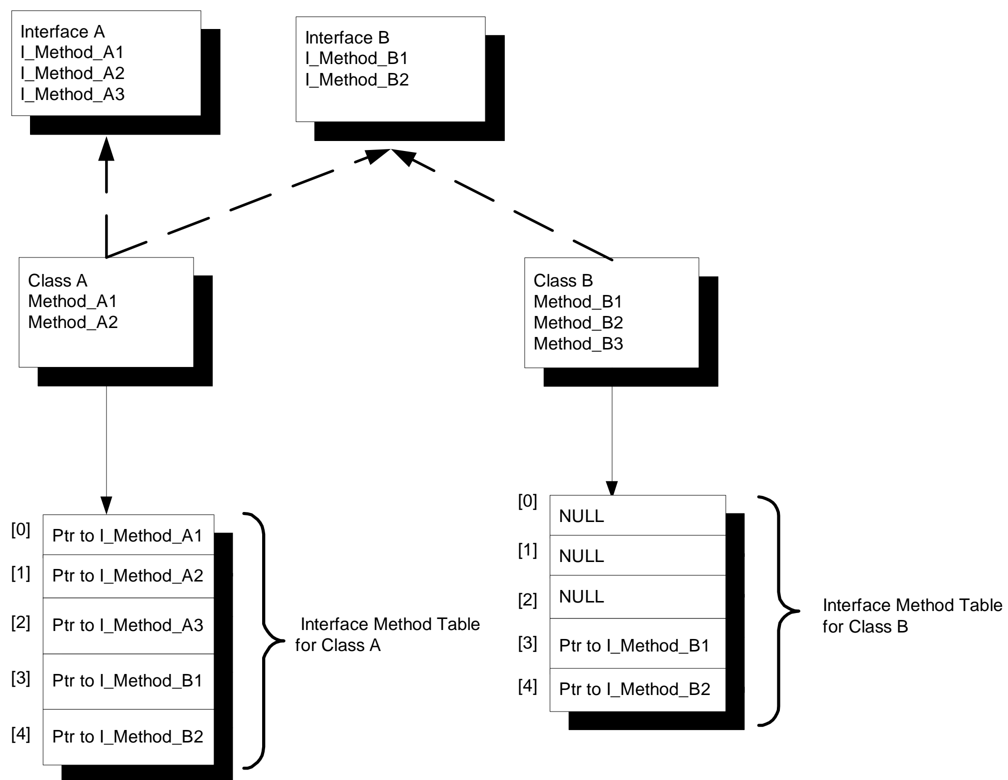

Figure 35 Deterministic interface method tables.

There are 2 classes in Figure 35, and since both implement interfaces, they both have an interface method table. Each interface method table allocates a unique entry for each interface method in the application. In Figure 35, entries 0 – 2 of the interface method

table for class A contain references to implementations of the methods declared in Interface A. It can be seen that entries 0 – 2 for Class B contain NULL entries as Class B does not implement Interface A. It can be observed that the size of the interface method table for each class is linear with the total number of interface method signatures in the application. For example, if an application has a total of 32 interfaces and each interface has 8 method signatures, then the interface method table belonging to each class will have 256 entries.

Another interface method Table organisation based on [SIE02] can be used to reduce the memory requirements of the scheme illustrated in Figure 36. This is at the expense of more complex runtime data structures and an additional level of indirection.

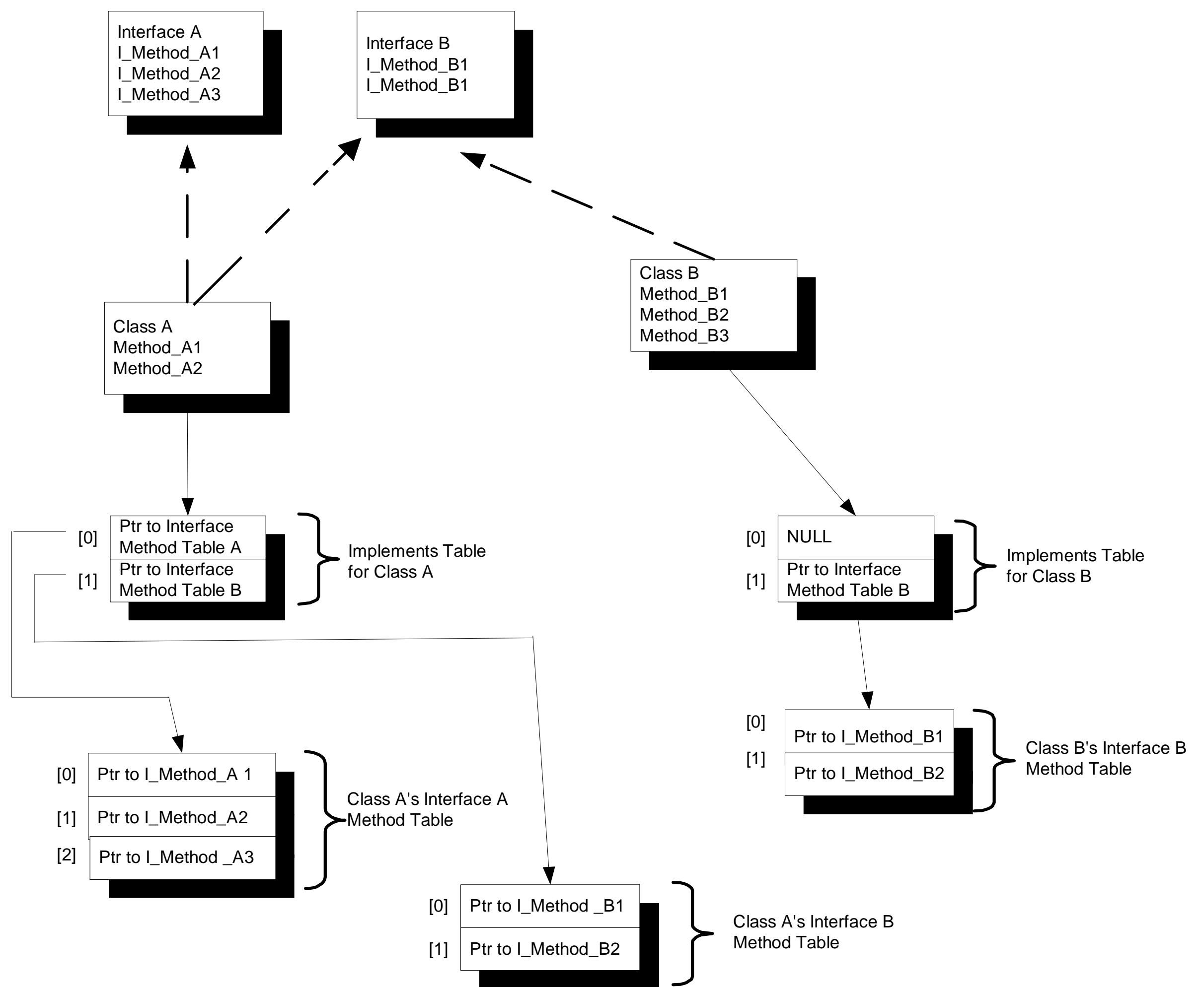

Figure 36 Deterministic Interface method tables.

Both classes A and B in Figure 36 have an additional runtime data structure called the *implements table*. The implements table has an entry for each interface in the application. The size of the implements table is the same for all classes in the system. Each class also has an interface method table for each interface implemented. The entries in the interface method tables contain references to the implemented methods in the class. It can be observed that the interface method tables do not contain any unused NULL entries as in Figure 35, however the implements table may include NULL entries. For example the implements table for Class B in Figure 36 contains an unused

entry at location 0, since Class B does not implement Interface A and because all implements tables throughout the application use this entry to refer to Interface A. It should be observed that the size of implements tables in Figure 36 is linear with the total number of interfaces in the application, instead of being linear with the number of interface methods as in Figure 35. Since an application is likely to contain fewer interfaces than interface methods because interfaces generally contain more than one method, this reduces the memory requirements.

Appendix 1 - Bytecode Execution Cycles

The following tables show the number of clock cycles to execute each JBC. The number of clock cycles for the base JVM is specified first, followed by JakHarta. If an instruction is not implemented, the entry is left blank. Details of the format and semantics of each instruction can be found in [VEN98].

Push Specific Constants	**Base JVM**	**JakHarta**
aconst_null		
iconst_m1	3	2
iconst_0	3	2
iconst_1	3	2
iconst_2	3	2
iconst_3	3	2
iconst_4	3	2
iconst_5	3	2
lconst_0		
lconst_1		
fconst_0		
fconst_1		
fconst_2		
dconst_0		
dconst_1		
Sign Extended and Push Literal	**Base JVM**	**JakHarta**
bipush		2
sipush		2
Push Item From Constant Pool	**Base JVM**	**JakHarta**
ldc		
ldc_w		
ldc2_w		

Push Item from Local Variable	Base JVM	JakHarta
iload	3	4
lload		
fload		
dload		
aload		4
iload_0	3	3
iload_1	3	3
iload_2	3	3
iload_3	3	3
lload_0		4
lload_1		4
lload_2		
lload_3		
fload_0		
fload_1		
fload_2		
fload_3		
dload_0		
dload_1		
dload_2		
dload_3		
aload_0		4
aload_1		4
aload_2		4
aload_3		4
Pop and Store in Local Variable	**Base JVM**	**JakHarta**
istore	2	3
lstore		
fstore		
dstore		
astore	2	3
istore_0	2	2
istore_1	2	2
istore_2	2	2
istore_3	2	2
lstore_0		
lstore_1		
lstore_2		
lstore_3		
fstore_0		
fstore_1		
fstore_2		
fstore_3		
dstore_0		
dstore_1		
dstore_2		
dstore_3		
astore_0	2	2
astore_1	2	2

astore_2	2	2
astore_3	2	2
Stack Manipulation	**Base JVM**	**JakHarta**
pop	1	1
pop2	1	1
dup	2	3
dup_x1		
dup_x2		
dup2		
dup2_x1		
dup2_x2		
swap	4	4
Push Item from Array	**Base JVM**	**JakHarta**
iaload		
laload		
faload		
daload		
aaload		
baload		
caload		
saload		
Pop and Store in Array	**Base JVM**	**JakHarta**
iastore		
lastore		
fastore		
dastore		
aastore		
bastore		
castore		
sastore		

Arithmetic and Logic	**Base JVM**	**JakHarta**
iadd	3	4
ladd		4
fadd		
dadd		
isub	3	4
lsub		
fsub		
dsub		
imul	3	4
lmul		
fmul		
dmul		
idiv		
ldiv		
fdiv		
ddiv		
irem		
lrem		
frem		
drem		
ineg	2	2
lneg		
fneg		
dneg		
ishl		
lshl		
ishr		
lshr		
iushr		
lushr		
iand	3	4
land		
ior	3	4
lor		
ixor	3	
lxor		
iinc	3	4

Conversions	**Base JVM**	**JakHarta**
i2l		
i2f		
i2d		
l2i		
l2f		
l2d		
f2i		
f2l		
f2d		
d2i		
d2l		
d2f		
i2b		
i2c		
i2s		
Flow Control	**Base JVM**	**JakHarta**
lcmp		
fcmpl		
fcmpg		
dcmpl		
dcmpg		
ifeq	2 or 3	3
ifne	2 or 3	3
iflt	2 or 3	3
ifge	2 or 3	3
ifgt	2 or 3	3
ifle	2 or 3	3
if_cmpeq	3	4
if_cmpne	3	4
if_cmplt	3	4
if_cmpge	3	4
if_cmpgt	3	4
if_cmple	3	4
if_acmpeq		
if_acmpne		
goto	3	3
jsr		
ret		
tableswitch		
lookupswitch		
ireturn		7
lreturn		
freturn		
dreturn		
areturn		
return		7

Operations on Classes/Objects	**Base JVM**	**JakHarta**
getstatic		9
putstatic		12
putfield		
getfield		
invokevirtual		
invokespecial		
invokestatic		21
invokeinterface		
new		
newarray		
anewarray		
arraylength		
athrow		
checkcast		
instanceof		
Miscellaneous	**Base JVM**	**JakHarta**
monitorenter		
monitorexit		
wide		
multianewarray		
ifnull		
ifnonnull		
goto_w		
jsr_w		
nop	1	1
Implementation Specific	**Base JVM**	**JakHarta**
pushsp		3
pushsb		3
pushrsp		3
pushpc		3
popsp		2
popsb		2
poprsp		2
poppc		2

Table 5 Bytecode Cycle Times for Base JVM and JakHarta.

References

[AND83] Andrews, G., and Schneider, F., 1983, "Concepts and Notations for Concurrent Programming", *ACM Computing Surveys*, 15(1), 3-44.

[AUD91] Audsley, N., C., Burns, A., Richardson, F., and Wellings, A. J., 1991, "Hard Real-Time Scheduling: The Deadline Monotonic Approach", Proceedings of the 8th IEEE Workshop on Real-Time Operating Systems and Software, Atlanta, Ga.

[BAC86] Bach, M., J., 1986, "The Design of the Unix Operating System", Prentice-Hall.

[BAR03] Barnes, J., 2003, "High Integrity Software: The Spark Approach", Addison Wesley.

[BAT01] Bate, I., Conmy, P., Kelly, T., and McDermid J., 2001, "Use of Modern Processors in Safety-Critical Applications", The Computer Journal, Vol. 44, No. 6, pp. 553.

[BEN82] Ben-Ari, M., 1982, "Principles of Concurrent Programming", Prentice Hall.

[BIE95] Bieman, J. M., and Zhao, J. X., 1995, "Reuse Through Inheritance: A Quantitative Study of C++ Software", ACM SIGSOFT Symposium on Software Reusability.

[BOO94] Booch, G., 1994, "Object-Oriented Analysis and Design", 2nd Ed., Addison-Wesley.

[BRI99] Briand, L. P., and Roy, D. m., 1999, “Meeting Deadlines in Hard Real-Time Systems: The Rate Monotonic Approach”, The Computer Society.

[BUR97] Burns, A., and Wellings, A., 1997, “Real Time Systems and Programming Languages”, Addison-Wesley.

[BUR98] Burns, A., and Wellings, A., 1998, “Concurrency in Ada”, Cambridge University Press.

[CEL02] Celoxica, 2002, “Handel-C Version 3.0 Language Reference Manual”, http://www.celoxica.com.

[CLA97] Cladingboel, C., 1997, “Hardware Compilation and the Java Virtual Machine”, MSc. Dissertation, Department of Computation, Oxford University.

[COA01A] Coates, G., 2001, “JVMs for Embedded Environments”, Java Developers Journal.

[COA01B] Coates, G., 2001, “J2ME Benchmarking: A Review”, Java Developers Journal.

[COA04] Coates, G., Goburdhun, A., and Green, P., N., 15 September 2004 “JakHarta: A Hardware Java Virtual Machine for Hard Real-Time Systems”, System-On-Chip Design, Test and Technology Postgraduate Seminar, Loughborough University.

[COA96] Coad, P and Mayfield, M, 1996, Java Design: Building Better Apps and Applets, Yourdon Press Computing Series.

[DEI90] Deitel, H., M., 1990, “Operating Systems”, 2nd Ed., Addison-Wesley.

[DER01] Derivation Systems, 2001, "LavaCore Configurable Java Processor Core", Xilinx and Derivation Systems.

[DIB02] Dibble, C., P., 2002, "Real-Time Java Platform Programming", Prentice Hall.

[DIJ69] Dijkstra, E. W., "Notes on Structured Programming", http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF

[ENG97] Engblom, J, 1997, Worst Case Execution Time Analysis for Optimized Code, MSc Thesis, Uppsala University.

[GAR75] Garey, M. R., and Johnson, D. S., 1975, "Complexity Results for Multiprocessor Scheduling Under Resource Constraints", 1975, Journal of Computing, Vol. 4 (4), pp. 397 – 411.

[GER03] German, A., 2003, "Software Static Code Analysis: Lessons Learned", QinetiQ Ltd.

[GOB04] Goburdhun, A., 2004, "Threading in a Hardware JVM", MPhil Thesis in preparation, Department of Computation, UMIST.

[GOS00] Gosling, J., Joy, B., Steele, G., Bracha, G., 2000, "The Java Language Specification".

[GOT02] Goth, G., 2002, "Has Object-Oriented Programming Delivered?", IEEE Software, Sep/Oct 2002.

[HAR] Hardin, D., S., aJile Systems: Low-Power Direct-Execution JavaTM Microprocessors for Real-Time and Networked Embedded Applications, aJile Systems, Inc., http://www.ajile.com/downloads/aJile-white-paper.pdf

[HAT95] Hatton, L., 1995, "Safer C: Developing Software for High Integrity and Safety-Critical Systems", McGraw-Hill.

[HEN96] Hennessy, J., and Patterson, A., 1996, "Computer Architecture: A Quantitative Approach", Morgan Kaufmann.

[HUN04] Hunt, J., 2004,The HIDOORS Methodology, Using Java in Real-Time and Embedded Systems, IST Project IST-2001-32329 HIDOORS, aicas gmbh.

[INM88] Occam 2 Reference Manual, 1988, Inmos Ltd, Prentice Hall.

[JCON] Real Time Java Core Extensions, J Consortium, www.j-consortium.org/rtwg/.

[JSR01] "Real-time Specification for Java", JSR-000001, https://rtsj.dev.java.net/.

[JSR133] Java Specification Request (JSR) 133, "The Java Memory Model", http://www.cs.umd.edu/~pugh/java/memoryModel.

[KAT84] Katevenis, M. G. H., 1984, "Reduced Instruction Set Computer Architectures for VLSI", ACM Doctoral Dissertation Award, The MIT Press.

[KLIG86] Kligerman, E., Stoyenko, A. D., 1986, "Real-Time Euclid: A Language for Reliable Real-Time Systems", IEEE Transactions on Software Engineering, Vol. SE-12, No. 9.

[KOO89] Koopman, P. Jr., 1989, "Stack Computers: the new wave", Mountain View Press.

[KOP97] Kopetz, H., 1997, “Real-Time Systems, Design Principles and Techniques for Distributed Embedded Applications”, Kluwer.

[KOR01] Koren, I., 2001, “Computer Arithmetic Algorithms”, A. K. Peters.

[KWO02] Kwon, J., Wellings, A., King, S., 2002, “Ravenscar-Java: A High Integrity Profile for Real-Time Java”, Department of Computer Science, University of York.

[LAP85] Laprie 1985, “Dependability computing and fault tolerance: Concepts and technology”, Digest of Papers, The Fifteenth Annual International Symposium on Fault Tolerant Computing, pp. 2-11.

[LEV03] Leveson, N., 2002, “Re: Object-Orientation vs. Safety-Critical”, Safety-Critical Mailing List, http:www.cs.york.ac.uk/hise/safety-critical-archive /2002/0203.html.

[LIN99] Lindholm, T., Yellin, F., 1999, “The Java Virtual Machine Specification”, Addison-Wesley Professional.

[LISK88] Liskov, B., 1988, “Data Abstraction and Hierarchy”, SIGPLAN Notices.

[LIU73] Liu, C. L., and Layland, J. W., 1973, “Scheduling Algorithms for Multi-Programming in a Hard Real-Time Environment”, Journal of the Association for Computing Machinery (ACM), Vol.20, No. 1, pp. 46-61.

[MAN98] Mangione C., 1998, “Performance Tests Show Java As Fast as C++”, Java World, www.javaworld.com/javaworld/ jw-02-1998/jw-02-jperf_p.html

[MAN01] Mano, M., M., and Kime, C., R., 2001, "Logic and Computer Design Fundamentals", Prentice Hall.

[MAY97] J Mayer, J and Downing, T, 1997, " The Java Virtual Machine", O'Reilly, http://cat.nyu.edu/~meyer/jasmin/.

[NAS04] NASA, 2004, "Handbook for Object-Oriented Technology in Aviation", Vol. 2: Considerations and Issues.

[NAZ] Nazomi Communications Inc., http://www.nazomi.com/.

[NIL05] Nilsen, K, 2004 – 2005, Aonix Inc., Personal communication.

[OPE01] The Open Group, http://www.opengroup.org.

[PUS89] Puschner, P., and Koza, C., 1989, "Calculating the Maximum Execution Time of Real-Time Programs", Real-Time Systems, Vol 1(2), pp. 159 –176.

[PUS02A] Puschner, P., Bernat, G., Wellings A., 2002, "Making Java Hard Real-Time", Annals of the Marie-Curie Fellowship Association, Vol. 2., Sect. 2.3.2.

[PUS02B] Puschner, P., Bernat, and Burns, A., 2002, "Writing Temporally Predictable Code", Proceedings of the Seventh International Workshop on Object-Oriented Real-Time Dependable System.

[PUS02C] Puschner, P., 2002, "Is Worst-Case Execution Analysis a Non-Problem? – Towards New Software and Hardware Architectures", 2nd Intl. Workshop on Worst Case Execution Time Analysis, http://www.cs.york.ac.uk/rts/wcet2002/paper/puschner.ps

[REX97] Erica Rex, 1997, “Baratz Tells the Java Success Story”, JavaWorld, http://java.sun.com/javaone/javaone97/java1-97-baratz.html.

[SCH92] Schnidt, W. J., 1992, “Issues in the design and implementation of a real – time garbage collection architecture”, An abstract of a Dissertation submitted to the Graduate faculty in partial fulfilment of the Degree of Doctor of Philosophy, Iowa State University.

[SEB01] Sebek, F., 2001, “The state of the art in Cache Memories and Real Time Systems”, MRTC Technical Report 01/37, Malardalen University, Sweden.

[SHA90] Sha, L., R., Rajkumar, and S., Sathaye, 1990, "Priority inheritance protocols: An approach to real-time synchronization." IEEE Transactions on Computers, 39 (9), 1175-1185: 1990.

[SIE02] Siebert , F., 2002, “Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages”, aicas gmbh.

[SIM99] Simon, D., E., 1999, “An Embedded Software Primer”, Addison Wesley.

[SUNA] Sun Microsystems, “Java Language Overview, Java 2 Platform”, http://java.sun.com/docs/overviews/java/java-overview-1.html .

[SUNB] Sun Microsystems, “Java 2 Platform Standard Edition (J2SE)”, http://java.sun.com/j2se/.

[SUNC] Sun Microsystems, “The K Virtual Machine”, White paper, http:java.sun.com/products/cldc/wp/.

[SUND] Sun Microsystems, “Java Native Interface Specification”, http://java.sun.com/j2se/1.4.2/docs/guide/jni/spec/jniTOC.html

[SUN96] Sun Microsystems, 1996, “PicoJava –I Microprocessor Core Architecture white paper”.

[TUR36] Turing, A., “On computable numbers, with an application to the Entscheidungs problem”, Proceedings of the London Mathematical Society, Series 2, 42 (1936), pp. 230-265.

[VEN98] Venners, B, 1998, “Inside the Java Virtual Machine”, McGraw-Hill.

[VUL03] Vulcan Machines, 2003, “MOON2 Java Processor”.

[XU93] Xu, J., Parnas, D. L., 1993, “On Satisfying Timing Constraints in Hard-Real-Time Systems”, IEEE Transactions on Software Engineering.

[YOR01] Safety Critical Email Group, Department of Computer Science, University of York, safety-critical@cs.york.ac.uk.