Low-Level Analysis of a Portable Java Byte Code WCET Analysis Framework

I. Bate, G. Bernat, G. Murphy
Department of Computer Science
University of York
York, YO10 5DD, UK
{ijb,bernat,greg}@cs.york.ac.uk

P. Puschner*
Institut für Technische Informatik
Technische Universität Wien
A-1040 Wien, Austria
peter@vmars.tuwien.ac.at

Abstract

To support portability, worst-case execution time (WCET) analysis of Java byte code is performed at two levels – machine-independent program-flow analysis at a higher level and machine-dependent timing analysis of individual program constructs at a lower level. This paper contributes a WCET analysis that computes worst-case execution frequencies of Java-Byte Codes within the software being analysed and accounts for platform dependent information, i.e., the processor's pipeline. The main part of the approach is platform independent, only a limited analysis is needed on a per-platform basis.

1 Introduction

Programs compiled into Java byte code (JBC) are highly portable, meaning that they provide the same functionality when executed on different (virtual) machines [7]. JBC is mostly generated from Java but it can also be generated from other languages like Ada. Due to its portability more and more people use and port JBC, even in application areas that it was not originally designed for. One of these areas is real-time programming.

In real-time programming it is not only the correct functionality of code that is crucial. The correct timing of code is equally important. To maintain the portability of real-time programs written in Java it is therefore not only important to conserve functional correctness, but also information about the program's timing. Timing information must be available in the target environment in order to ensure that the given program will meet its timing constraints on the (virtual) machine on which it is intended to run.

In a previous paper we presented an approach for a portable Worst-Case Execution Time (WCET) analysis for JBC programs generated from Java or other programming languages [1]. This approach characterizes those properties of a program that determine its WCET in a machine-independent, abstract way. Although the timing information is abstract, it represents the details that are necessary to port the information to a specific target platform and take into account its features, e.g., instruction pipeline or cache. After completion of the machine-independent WCET analysis, the WCET information is added to the Java Class File (JCF) of the program, which also holds the program's JBC. Whenever the program is to be ported to a specific machine, the WCET information contained in the JCF is augmented with machine-specific information and evaluated for this machine. This evaluation checks that the program completes on the given platform within the CPU time that has been allocated to it.

The presented approach to a low-level WCET analysis for Java byte code has the following properties:

- The technique is portable to different targets and different types of JBC execution: interpretation, ahead-of-time compilation, and direct execution on hardware.
- Porting the WCET information to a specific target does not require the availability of complex tools. The target specific analysis requires only a small amount of resources and could even be implemented as part of a virtual machine.
- Taking into account hardware characteristics in the analysis allows us to keep the pessimism of the WCET analysis within reasonable bounds.
- Analysis of JBC without relying on source code. All information for the analysis is extracted from the JCF.

The contribution of this paper is an approach to portable low-level WCET analysis. We demonstrate how WCET information can be used to exploit knowledge about the pipeline of processors to obtain a portable WCET analysis of high quality. Other hardware features, e.g., cache, are not considered in this paper, but will be covered in future work.

This paper builds on previous work [1] which shows that a high-level WCET analysis of JBC is possible with-

*The presented work was performed while the last author was on a research stay at the University of York. This stay was supported by the EU Marie-Curie Fellowship, contract no. HPMF-CT-1999-00184.

out needing to know the source code from which it was created. The previous work [1] deals with the principal analyzability and assumed a constant, hardware-independent execution time for each byte code. This paper extends the previous work by using information about hardware-specific features, in particular instruction pipelines, to improve the quality of portable WCET analysis.

Techniques for the WCET analysis of caches and pipelines have been presented before [4, 5, 6]. Rather than modeling the hardware features of a specific processor the focus of this work is on making the analysis of these hardware-specific features as portable as possible. The paper shows which types of information and which computation methods are necessary to reach this aim.

The computation method uses simple calculation rules as used for the WCET calculation of simple computer architectures in [10, 11], and later extended for the modeling of instruction pipelines, [6]. While these approaches operate on hardware-specific instruction execution times from the very first step, the analysis presented here manipulates instruction frequencies. Dealing with instruction frequencies requires a modification of the computation schema and serves our central goal of portability.

Section 2 gives an overview of the previous work on the high-level WCET analysis of JBC. Section 3 introduces the virtual-machine timing model used throughout the paper. Section 4 presents instruction frequency vectors – vectors that keep information about the number of occurrences of instructions in JBC – and introduces the rules for their manipulation. Section 5 shows how machine-independent and machine-dependent pipeline information are combined to compute the WCET of JBC for a particular (virtual) machine. An example and an evaluation of our approach are presented in Section 6. Section 7 concludes the paper.

2 High-Level WCET analysis of JBC

2.1 Java Byte Code

JBC is the object-code format of a stack based Virtual Machine (VM). A program (in Java, Ada or any other language) is compiled into a JCF which holds the definitions of constants methods, attributes and other information as well as the actual JBC. The Java VM is the name of an engine which is able to execute JBC programs. The portability is achieved by producing an appropriate VM for each different platform. For details on JBC and JCFs see [7].

There are four main types of VM implementation:

- Interpreted: The JVM is an executive that interprets each byte code and executes it accordingly.
- Ahead of Time Translator (AOT): All code is translated to machine code before execution (e.g., at boot-up).
- Just-in-time compiler (JIT): The first time a method is called, it is translated into object code, subsequent calls benefit from code already in machine code format.
- Hardware VM: a processor that implements the JBC instruction set.

JIT provides good average-case performance but it is unsuitable for real-time systems as the worst-case computation time has to include the time to compile the code which may be large and difficult to predict. The predictability and portability of AOT based implementations depends on the level of optimisation performed during translation. If complex optimisations are performed, then analysis of the WCET needs to be done in detail on the generated object code and therefore it is not portable. Instead a simpler translation process is assumed in which each JBC instruction is translated into the corresponding object code.

The models which are predictable and portable and are therefore interesting from the perspective of analysis are: the hardware VM, the simple AOT translation and the interpreted VM.

Note that there are other issues in the implementation of the VM that affect its performance, for instance the memory model and the garbage collection mechanism. A technique suitable for real-time systems (like incremental garbage collection, or simply non-relocatable memory [2, 3]) is assumed. In any case their effects are accounted for in the schedulability analysis, not in the WCET analysis.

2.2 High-Level WCET Analysis

In a previous paper [1] it is shown how high-level analysis can be performed on JBC alone without relying on the original source code. This is due to the fact that the Java Class Verifier checks the correctness of the object code of JCFs (see [7]). Two features of JCFs are of special importance for WCET analysis. First, all branch instructions in the JBC jump to known labels (i.e., it is forbidden to jump to an address that is held in a variable). Second, for each location in the JBC it is possible to predict the configuration of the stack off-line.

The first property allows the basic blocks and the control flow graph to be extracted, the second allows data-flow analysis to be performed. Using standard compiler techniques [8] it is possible to detect loops, loop bounds, induction variables, ranges of variables etc.

2.3 Portable WCET Annotations

No matter how accurate the high-level analysis is, there are constructs for which automatic tools cannot determine the information required for WCET analysis. For instance, in the general case it may not be possible to determine the maximum number of iterations of a loop. This and other complex code structures need the support of annotations to aid the WCET analysis process.

WCET annotations are usually introduced as specially formatted comments. This approach is not feasible for JBC as it requires support from the compiler to translate these comments into the JCF. Moreover, the notation syntax is likely to depend on the source language. For this reason annotations are inserted into the code as calls to the predefined WCETAn class. These special calls, which have no functional effect, are extracted from the corresponding JCF and used by the WCET tool.

The WCETAn class is language-independent, see [1]. It allows scopes or code-block boundaries, maximum number of iterations of loops, execution modes and arbitrary path execution frequencies to be described. [9].

Portability is achieved by mapping this high-level information onto the low-level timing model of the VM. The analysis is portable from the high-level point of view as it does not depend on a source language or compiler, it performs all high-level analysis at the JBC level. The next stage is to provide a mechanism to combine this high-level analysis with the low-level timing features of the VM.

3 Virtual Machine Timing Issues

This section is to define the timing information relevant to WCET analysis for each VM implementation. The basic approach is to provide the WCET of each JBC instruction. The WCET of a section of is obtained from the WCET of each basic block of the control flow graph (by adding the timing of the instructions within it) by collapsing the corresponding graph. Unfortunately, the approach is not as simple as it may seem. There are two problems that need to be addressed.

First, there are JBC instructions which have non-predictable execution time, or in the worst case, a high upper bound on the execution time (for example athrow and checkcast). The main problem is that the behaviour of these instructions (and therefore their execution times) depend on the context in which they are executed. More precisely, they depend on the hierarchy of classes of their call. This is because the implementation needs to scan the hierarchy of classes until it finds the matching handler or object. Although this can be quite expensive, the height of the tower of classes is bounded and can be determined off-line. Another example are operations on multidimensional arrays in which the WCET depends on the number of dimensions of the array and method invocation where the cost of the call depends on the number of arguments of the call. The solution of this problem requires expressing the timing of each instruction as a function of a parameter n, the number of iterations, we call it the *internal iteration factor*. For simplicity only a linear combination is considered, i.e. the WCET of an instruction is a function of n. It is given by $a + bn$ where a is the execution time for the first iteration and b is the cost to perform each additional iteration.

Second, there are instructions which make the code unpredictable, e.g., the invokevirtual method for calling virtual methods. In the general Java environment, this method may request the method to be retrieved from a remote host. This is clearly unacceptable in a system that needs bounded worst-case execution times. It is assumed that either calls to virtual methods are not allowed in code segments on which WCET analysis has to be performed (in the same way that unbounded loops are not allowed), or that the method is always prefetched, considering it as a special case of method invocation (with the additional bounded cost of selecting which method to invoke).

Apart from these two sources of unpredictability the instructions have fairly constant execution times in all VM implementations considered (interpreted, simple AOT translator and HW VM). This allows us to build a table with the WCETs of the individual JBC instructions. This table is called the VM Timing model.

3.1 Special JBC instructions

For some JBC instructions the execution times are potentially variable. Obviously, both the execution strategies and the implementation method has a major impact upon this. The implementation of long (i.e., 64-bit) logical and arithmetic operations on the 32-bit integer unit of the Motorola M68040 may require a number of iterations. However, if the operation were to be carried out on the floating point unit, the time complexity would be constant because the floating point unit performs 80 bit wide operations of which 64 bits are for the signed mantissa. Similarly, the time complexity of the tableswitch and lookupswitch byte codes may or may not be constant. In an AOT compiler, the correct branch may be pre-selected whereas an interpreter may need to search through the set of entries provided as a variable number of arguments.

The invokevirtual, invokeinterface and athrow byte codes require the hierarchy of classes to be searched with the latter also needing the set of handler entries for each class to be traversed. Analysis can, however, determine an upper bound for these searches. It is noted that the run-time type-identification byte codes checkcast and instanceof can easily be implemented with constant time complexity.

The execution time of the multianewarray byte code depends on the number of dimensions of the array being created. Moreover, in certain implementations the execution time also depends on the number of elements in the array. Again, the computation of an upper bound is straightforward.

3.2 VM Timing Model

A VM Timing model for a particular combination of VM and processor is a data structure holding information about the WCET of each JBC. $T(\mathsf{p}) = (x, y)$ denotes the WCET of JBC instruction p. Note that $T(\mathsf{p})$ is of the form $x + yn$ where x represents the time it takes to execute the instruction, and y the additional time taken to execute each internal

Table 1. Most Frequent JBC Pairs

Pair	Frequency %
aload_0, getfield	9.93
dup, bipush	3.91
nnew, dup	3.66
bastore, dup	2.79
putfield, aload_0	2.04

iteration (for example, the number of elements to push on the stack for a method invocation). Therefore, $x + yn$ gives the WCET to execute n internal iterations. The parameter y is only used for JBC instructions that iterate internally. For most of the instructions $y = 0$.

Also, as discussed later in section 5, the "gain time" due to pipeline effects across k JBC instructions is considered. For a sequence of instructions p_1;p_2;...;p_k the gain factor is written as $\lambda(\mathsf{p}_1, \mathsf{p}_2, ..., \mathsf{p}_k)$. It is infeasible to provide the gain factors for all possible sequences of instructions. Depending on the VM and processor characteristics, only a subset of the gain factors of the relevant groups of instructions shall be considered. For simplicity purposes, λ is defined for instructions with constant execution time (instructions where $y = 0$).

In the rest of the paper only the gain factors of pairs of JBC instructions are considered. Experiments showed that taking into account the pipeline effects across more than two instructions improves the WCET-analysis only marginally. The study also revealed that even a substantial set of JBC pairs hardly ever occur. A small set of JBC pairs is sufficient to characterize most of the byte-code combinations occuring in compiled code (see Table 1 for the five most frequent JBC pairs). This suggests that only a subset of the JBC subsequences need to be represented in the tables of a WCET tool for JBC.

Section 6 presents an example of the VM timing tables of JBC for the PowerPC processor and an internal AOT translator. Note that this timing model is accurate for VM implementations that do not optimise during code generation. For AOT compilers that perform optimisation on the object code this approach is applicable but pessimistic. If tighter bounds are required in such scenarios, then the generated object code for each platform in which the program may run needs to be analysed. This approach is clearly non-portable and therefore not considered further in this paper.

In summary, the timing model of a VM is a pair of tables T and λ that list the WCET of each JBC instruction and the gain factors due to the favorable effects of pipelines across instruction sequences.

4 Worst-Case Execution Frequency Vectors

The next step is to map the high-level WCET analysis with the low-level timing model of the VM. There are two different ways to do this.

The traditional one is to use the high-level control-flow graph and annotate each JBC instruction with the information of the VM timing model, thus obtaining the WCET of each basic block. The control-flow graph is then collapsed and the WCET of the section of code obtained. The drawback of this approach is that it requires the timing information of each JBC instruction which may not be available yet.

For this reason instead of collapsing the graph after each basic block has been annotated with the timing information, the graph is first collapsed to obtain not the worst-case execution time but the worst-case execution frequency (WCEF) of each JBC instruction. With the information on the WCEF of instructions, it is then easy to combine it later with the VM timing model to obtain the WCET.

The process is formally described in the rest of this section. Let B be the set of JBC instructions and F_S a vector of execution frequencies of each of the instructions in B for a given code segment S. $F_S(p) = (a, b)$ where a is the maximum number of times that JBC instruction $p \in B$ is executed in the scope S and b is the maximum number of internal iterations.

The WCEF vectors can be built in a bottom-up fashion with an approach similar to the timing schema [11]:

- Sequential: For two sequential segments of code S1;S2 with WCEF F_{S1} and F_{S2} the WCEF of the sequence S=S1;S2 is:

$$F_S = F_{S1} + F_{S2} \tag{1}$$

  where the addition of vectors has the usual meaning. Note that the addition of pairs $(a_1, b_1) + (a_2, b_2) = (a_1 + a_2, b_1 + b_2)$ has the expected meaning. In the worst case the instruction is executed $a_1 + a_2$ times and iterates $b_1 + b_2$ times.

- Iteration: For a loop S with header H, body B, and a maximum number of iterations n the WCEF is:

$$F_S = (n + 1)F_H + nF_B \tag{2}$$

  where the addition of vectors and the multiplication of a vector by a pair (a, b) have the usual meaning.

- Conditional: For two alternative branches $S1$ and $S2$ of a conditional statement S the WCEF of S is:

$$F_S(p) = \max\{F_{S1}(p), F_{S2}(p)\} \;\; \forall p \in B \tag{3}$$

  where $\max\{(a_1, b_1), (a_2, b_2)\} = (\max\{a_1, a_2\}, \max\{b_1, b_2\})$

Note that this operation is pessimistic but safe. Consider the following example of an if-then-else with only two different JBC instructions x, y. Assume the then branch has a WCEF vector of $F_{\text{then}}(x) = (1,0)$ and $F_{\text{then}}(y) = (10,0)$] and the else branch has a WCEF of $F_{\text{else}}(x) = (12,0)$ and $F_{\text{else}}(y) = (3,0)$. The WCET will depend on the WCET of a given code section of each JBC, but this information is unknown, the only safe approach is to consider the maximum of each of the instructions in turn, which results in $F_{\text{then-else}} < x, y >=< (12,0),(10,0) >$. This is pessimistic, because if we assume, for instance, that both JBC take 1 cycle to execute, then the if part would take 11 cycles, the else part would take 15 cycles and therefore the WCET of the whole code would be 15. However the WCET after the fusion leads to $12 + 10 = 22$ cycles.

Given a WCEF vector F_S and a VM timing table T then the WCET of the scope S is:

$$WCET_S = \sum_{\forall p \in B} F_S(p).a\, T(p).x + F_S(p).b\, T(p).y \quad (4)$$

where T has entries $T(p) = (x, y)$.

This assumes each JBC instruction executes independently of the others.

5 Low-Level Timing Analysis of JBC

So far the model of the higher-level path analysis of JBC and how the JBC is to be handled on a specific platform by the JVM has been presented. The purpose of this section is to expand the higher-level analysis to account for the overheads of the JVM and the speed-up associated with the processor's pipeline mechanism leading to an overall approach to timing analysis.

5.1 Overheads Due to the Java Virtual Machine

The overheads being considered are those associated with the translation of JBCs during the software's execution. From section 3, the assumed execution model is that JBCs are individually translated by either software or hardware, i.e., there is no optimisation across multiple JBCs. The fact the JBCs are translated means that each JBC's execution time has to be adapted as shown in equation (5) to account for the overheads associated with translation/interpretation. The overheads are present independent of whether the JBC is translated by hardware or software but the method of translation does affect the overheads' characteristics, e.g., their magnitude.

$$C_i' = C_i + \Delta_i \quad (5)$$

where i identifies a particular JBC. C_i is the worst-case execution of the JBC without any overheads. Δ_i is the overhead associated with translating the particular JBC and C_i' is the modified worst-case execution of the JBC.

A characteristic of the JVMs presented in section 3 is that most instructions have a constant overhead, i.e., $\Delta_i = \Delta$. For the AOT approach, $\Delta_i = 0$ leading to $C_i' = C_i$. For the remainder of the paper only the overall execution time C_i' associated with the interpretation of the instruction is considered and not its component parts.

5.2 Accounting for Pipeline Mechanisms

A significant problem with the approach presented so far is the pessimism that can arise because the analysis does not account for local effects, e.g., caches and pipelines.

In [6], a method of performing pipeline analysis based on reservation tables is described. Table 2 illustrates how the technique represents the execution of a JBC on a specific platform. The table shows the JBC being executed as three instructions (labeled 1, 2, 3) on a platform with a four-stage pipeline (fetch, dispatch, execute, complete) where the execute stage has three units (SRU, FPU, IU) that can be used concurrently. Along the x-axis of the table is the current clock cycle and on the y-axis the processing resource, i.e., pipeline stage. The numbers at the intersection points of the coordinate system identify the instructions being analysed, i.e., at time 4 instruction "3" is dispatched and instruction "2" is executing in the Floating Point Unit (FPU). The reservation table identifies the resources that are available at a particular time so that the execution pattern of the next instruction can be determined. To simplify the example, the pipeline mechanism is assumed to execute in-order to avoid the timing anomalies due to dynamically scheduled pipelines. Hence there is a one cycle gap between the execution of instruction "3" and its completion – instruction "3" is not allowed to complete before instruction "2". In practice, many modern processors would actually complete both instructions in a single clock cycle, i.e., in cycle 6.

Table 2. Example Reservation Table for a JBC that Executes in Three CPU Instructions

	1	2	3	4	5	6	7
Fetch	1	2	3				
Dispatch		1	2	3			
Execute - LSU			1				
Execute - FPU				2	2		
Execute - IU					3		
Complete				1		2	3

It is proposed that the pipeline speed-up is allowed for by producing reservation tables for each JBC (including translation) and then accounting for consecutively executing JBC by concatenating each JBCs' reservation table. Figure 1 gives a high-level overview of how a stream including the associated overheads of JBCs is handled.

The concatenation of two reservation tables is shown in more detail in Table 3. In this case the reservation tables are identical each being the same as the one in Table 2. The presence of the second table is emphasised by the use of **bold**.

Figure 1. The Concatenation of a Stream of JBCs Each With An Overhead Associated

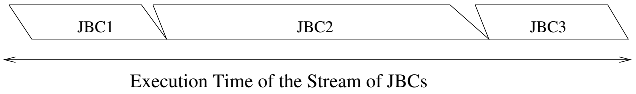

Table 3. The Concatenation of Two Reservation Tables

	1	2	3	4	5	6	7	8	9	10	11
Fetch	1	2	3		**1**	**2**	**3**				
Dispatch		1	2	3		**1**	**2**	**3**			
LSU			1				**1**				
FPU				2	2			**2**	**2**		
IU					3				**3**		
Complete				1		2	3	**1**		**2**	**3**

Table 3 shows that the two reservation tables are concatenated by pushing the tables together and determining the earliest point at which the two tables would not have a resource conflict. In this example, the resource conflict in question is in the completion stage of the pipeline where the second instance of instruction "1" completes the cycle after the first instance of instruction "3". The method of pushing the tables together until the cycle before a resource conflict occurs is for analysis purposes only. In practice the fetch of the second instance of instruction "1" would be the clock cycle after the fetch of the first instance of instruction "3". However analysing the pipeline mechanisms in the same way as it executes in practice would require the second reservation table to be re-analysed, whereas this approach does not. Concatenating the reservation tables reduces the pessimism by 3 clock cycles because, if we treat the two reservation tables independently then the overall execution time is $2 * 7$ clock cycles compared to 11 clock cycles resulting from the concatenation.

In [4] it is shown how timing analysis, such as that shown in equation (5), can be improved to account for the speed-up due to pipeline analysis. Equation (6) shows how the pipeline effect between the translation and the related JBC can be accounted for.

$$C'_i + C'_j - \lambda(C'_i, C'_j) \qquad (6)$$

where $\lambda(X, Y)$ is the speed-up associated with allowing for the pipeline mechanism.

5.3 The Integrated Analysis

For descriptive purposes it is assumed that the pipeline effects of all pairs of instructions are available.

Equation 4 can be now rewritten accounting for the pipeline effects as shown in equation 7. Let S^2 be the set of significant pairs of consecutive instructions in code segment S. In a first approach, it only considers sequential pairs of instructions. That is, pairs that start with a branch instruction and subroutine calls are not considered. Note that the same execution-frequency schema presented in section 4 can be applied for the WCEF of pairs of instructions. Therefore it can be assumed that $F_S(\mathsf{p}, \mathsf{q})$, which is the WCEF of the pair $\mathsf{p}; \mathsf{q}$, is available for each p and q. Then the WCET is given by:

$$\begin{aligned} WCET_S = \sum_{\forall p \in B} (F_S(p).aT(p).x + \\ + F_S(p).bT(p).y) - \\ - \sum_{\forall (p,q) \in S^2} F_S(p,q)\lambda(p,q) \end{aligned} \qquad (7)$$

Taking the approach of allowing for pipeline effects between consecutive instructions means N^2 speed-up effects would be needed, where N is the number of JBCs. However the actual number would be reduced to $N * M$, where M is the number of different JBC combinations that are occurring in the code.

Figure 2. Bubble Sort Code

```
public static void bubble(int a[])
{ int i, j, t;
  int size = 10;
  for (i=size-1; i>0; i--)
      /* max. iterations: size-1 */
  { for (j=1; j<=i; j++)
      /* max. iterations: size-1 */
    { /* max. exec. in bubble:   */
      /*    size*(size-1)/2      */
      if (a[j-1] > a[j])
      { /* max. exec. in bubble:*/
        /*    size*(size-1)/2   */
        t      = a[j-1];
        a[j-1] = a[j];
        a[j]   = t;
      }
    }
  }
}
```

6 Evaluation

The complete analysis is illustrated with the example code of the Bubble Sort algorithm presented in Figure 2. The code shows the maximum number of iterations of the inner and outer loops in comments. Figure 3 shows the graph of basic blocks. For this example only 14 different JBCs are generated by the SUN JDK 1.2. To simplify the exposition we assume that the WCET of all of the instructions do not have internal iteration factors. Table 4 shows the WCEF of each basic block and the final WCEF of the whole code section. The WCEF is obtained as follows:

$$B1+10B8+9(B2+B7)+46B6+45(B3+B4+B5)+B9 \quad (8)$$

Table 5 shows the execution frequencies of the relevant pairs of JBCs. This table is used for computing the gain due to the pipeline.

Figure 3. Graph of Basic Blocks of Bubble Sort

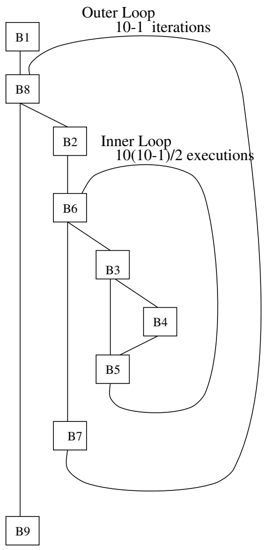

Table 4. WCEFs of the Individual Basic Blocks and the Whole Program Assuming $size = 10$

	B1	B2	B3	B4	B5	B6	B7	B8	WCEF
ALOAD_0	0	0	2	4	0	0	0	0	270
BIPUSH	1	0	0	0	0	0	0	0	1
GO_TO	1	1	0	0	0	0	0	0	10
IALOAD	0	0	2	3	0	0	0	0	225
IASTORE	0	0	0	2	0	0	0	0	90
ICONST_0	0	0	1	0	0	0	0	0	45
ICONST_1	0	1	0	2	0	0	0	0	99
IF_ICMPLE	0	0	1	0	0	1	0	0	91
IFGT	0	0	0	0	0	0	0	1	10
IINC	0	0	0	0	1	0	1	0	54
ILOAD_1	0	0	0	0	0	1	0	1	56
ISTORE_1	1	0	0	0	0	0	0	0	1
ISUB	0	0	1	2	0	0	0	0	135

6.1 Timing model for the PowerPC

The analysis is performed for an internal JBC AOT translator which does not perform optimisations across JBCs and generates code for the PowerPC processor. The PowerPC was considered because the pipeline effects are very significant. In this version the Java operand stack is implemented in memory. The example in Figure **??** shows the translation of the isub byte code. The respective pipeline analysis is shown in Figure 5.

Table 5. WCEFs of the Significant Pairs of Instructions (B5 and B7 are empty and have been omitted)

JBC pairs		B1	B2	B3	B4	B6	B8	WCEF
aload_0	iload_2			2	4			270
bipush	istore_1	1						1
iaload	iastore				1			45
iaload	if_icmple			1				45
iaload	istore_3				1			45
iastore	aload_0			1	1			90
iconst_1	istore_2		1					9
iconst_1	isub			1	2			135
iload_1	if_icmple					1		46
iload_1	ifgt						1	10
iload_2	iaload			1	1			90
iload_2	iconst_1			1	2			135
iload_2	iload_1					1		46
iload_2	iload_3				1			45
iload_3	iastore				1			45
istore_1	goto	1						1
istore_2	goto		1					9
istore_3	aload_0				1			45
isub	aload_0				1			45
isub	iaload			1	1			90

Figure 4. PowerPC Code of the isub **JBC.**

```
isub
(1) lwz  r28,-4(r5) |load top of the stack
                    | r5=top of the stack
(2) lwz  r27,-8(r5) |load 2nd element
(3) sub  r27,r27,r28|perform subtraction
(4) stw  r27,-8(r5) |store the result
(5) addi r5,r5,-4   |adjust top of stack
```

Figure 5. Pipeline Analysis of the isub **Instruction. Total WCET = 10 Cycles.**

	1	2	3	4	5	6	7	8	9	10
Fetch	1	2	3	4	5					
Dispatch		1	2	3	4	5				
LSU			1	1	2	2	4	4	5	
IU					3					
Complete					1		2	3	4	5

A summary of the timing model of the instructions generated for the Bubble Sort example is shown in Table 6. As mentioned above, this timing model assumes in-order execution. The table shows the WCET of the JBC instructions (in machine cycles). These values already include the intra-JBC pipeline effects. Finally, Table 7 shows the inter-JBC pipeline gain factors for the pairs of instructions that appear in the Bubble Sort example.

Based on the information obtained, it is possible to evaluate the WCET of the code. The WCET without considering the inter-JBC pipeline effect is 8808 machine cycles. The difference between the WCET with and without consideration of pipeline effects equates to 3886. Therefore the WCET including the inter-JBC pipeline effect is 4992. Accounting for pipeline effects reduces the pessimism of the

Table 6. Timing Model of the PowerPC Processor (in Machine Cycles). Only Instructions Used by our Bubble Sort are Shown.

JBC	T(p)	JBC	T(p)
aload	5	ifgt	6
bipush	7	goto	2
go_to	2	iinc	4
iaload	12	iload	6
iastore	13	istore	6
iconst_0	9	isub	10
if_icmple	9		

analysis by $\frac{3886}{8808} \times 100 = 44.4\%$.

Table 7. Inter-JBC Pipeline Gain Factors

JBC		Gain	JBC		Gain
aload_0	iload_2	3	iload_1	if_icmple	3
bipush	istore_1	3	iload_1	ifgt	3
iaload	aload_0	3	iload_2	iaload	3
iaload	iastore	3	iload_2	iconst_1	3
iaload	if_icmple	3	iload_2	iload_1	3
iaload	istore_3	3	iload_2	iload_3	3
iastore	aload_0	3	iload_3	iastore	3
iastore	iinc	3	istore_1	goto	4
iconst_1	istore_2	3	istore_2	goto	4
iconst_1	isub	3	istore_3	aload_0	3
iinc	iload_1	3	isub	aload_0	3
iinc	iload_2	3	isub	iaload	3

7 Conclusion

In this paper an approach has been presented for a portable worst-case execution time analysis for JBC programs, generated from Java and other programming languages like Ada. This approach characterizes those properties of a program that determine its WCET in a machine independent, abstract way. Although the timing information is abstract it represents the details that are necessary to port the information to a specific target platform and take into account its features, e.g., instruction pipeline or cache.

This timing model is mapped with the worst-case execution frequency vector which accounts for the maximum number of executions of each possible JBC. Pipeline effects across JBCs are accounted for by computing the maximum frequency of pairs of instructions.

Future issues to investigated include the analysis of the subset of the most frequent pairs, compression of frequency information by partitioning the JBC into classes of similar instructions, and the analysis of the complexity of evaluation on the target for portability purposes.

Acknowledgement

The authors would like to thank Mike Bennett for his help on producing the PowerPC code. This work has been partially funded by the DERA with project CSM/1254.

References

[1] G. Bernat, A. Burns, and A. Wellings. Portable worst case execution time analysis using java byte code. In *proc. 6th EUROMICRO conference on Real-time Systems*, June 2000.

[2] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. *Real-time Specification for Java*. Addisson Wesley, 2000.

[3] J Consortium real-time Java working group. Real-time core extensions for the Java platform (draft), 2000. http:\\www.j-consortium.org.

[4] J. Engblom and A. Ermedahl. Pipeline timing analysis using a trace-driven simulator. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, pages 88–95, December 1999.

[5] C. A. Healy, R.D. Arnold, F. Mueller, D.B. Whalley, and M.G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1):53–70, 1999.

[6] S. Lim, Y. Bae, G. Jang, B. Rhee, S. Min, C. Park, H. Shin, K. Park, and C. Kim. An accurate worst case timing analysis for RISC processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, July 1995.

[7] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addisson Wesley, second edition, 1999.

[8] S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers, 1997.

[9] P. Puschner and A. Schedl. Computing maximum task execution times - a graph based approach. *Real-Time Systems*, 13(1):67–91, July 1997.

[10] P. Puschner and Ch. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems Journal*, 1(2), 1989.

[11] A. Shaw. Reasoning about time in high-level language software. *IEEE Transactions on Software Engineering*, 17(7):875–889, 1989.