Java Virtual-Machine Support for Portable Worst-Case Execution-Time Analysis

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

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

Abstract

The current trend towards the usage of Java in real-time, supported by two specifications (Real-Time Java and Real-Time Core extensions for the Java platform) requires adequate schedulability analysis, and consequently, worst case execution time (WCET) analysis techniques for the Java platform. This paper proposes a framework for providing portable WCET analysis for the Java platform. Portability means that the analysis is language and hardware independent. It is achieved by separating the WCET analysis process in three stages and by analysing the Java Byte Code, not the high-level source code, thus enabling the analysis of programs written in other languages (such as Ada and compiled for the Java virtual machine). The three stages are: a Java virtual machine platform dependent (low-level) analysis, a software dependent (high-level) analysis and an on-line integration step.

1 Introduction

To determine whether the deadlines of a real-time system can be met by a schedulability analysis technique it is necessary to compute the Worst-Case Execution Time (WCET), i.e., the longest execution time of particular sections of code. Timing estimates should be based on analysis, rather than test, because in general it is not possible to ascertain that the tests have really found the longest execution time. *WCET analysis* computes these execution-time bounds by considering the maximum number of executions of all parts of the code and the maximum execution times of each of the parts of code for all possible program paths. Note that the latter information depends on the characteristics of the hardware on which the code is expected to run.

Current WCET techniques are tied to a particular language, target architecture, and even compiler optimizations. The Java framework has been conceived with portability as one of its driving design goals. The emphasis has been on functional portability not on temporal portability. The execution time of a Java program on a different Java Virtual Machine (JVM) is not known. This is a serious problem for the approaches that want to use Java in real-time contexts. There are currently two ongoing specification processes for providing real-time extensions to Java. Real-Time Java (RTJ) [1] and the Real-time core extensions for the Java platform [2]. These specifications have addressed the issues related to using Java in a real-time context including execution time analysis, scheduling support, memory management issues, interaction between non-real-time Java and real-time Java programs, device management among others.

We acknowledge the contributions made by both specifications towards a real-time enabled Java platform. However, we argue that they do not provide a satisfactory solution for portable WCET analysis. RTJ requires the user to provide a `cost` parameter. Core allows us to compute the WCET of a section of code, but relies on the ability to perform automated complex code analysis by the VM at run-time because no support for user annotations is provided. Although this is adequate for small sections of code it has been shown that the amount of pessimism included by automated analysis techniques can be extremely large (experiments have reported between one and two orders of magnitude of pessimism [3]). Sources of this pessimism include the inability to detect mutually exclusive paths and variable nested loop bounds among others.

We envisage a scenario where users can download their components to a Java Virtual Machine and inquire whether their components would "fit" within their deadline. This requires the ability to deliver a code with timing information and timing requirements, and to interrogate the VM of their temporal performance. Another possible scenario is when code migration needs to be performed in a distributed real-time systems. A schedulable entity could only be migrated to a different node in the system if the node would remain schedulable. A protocol for code migration needs to be able to determine the execution time of a particular code segment in a remote node.

Our work concentrates on providing a portable WCET execution time analysis framework for the Java platform. Programs written within this framework can be analyzed for their WCET in a portable way and the timely operation of code can easily be verified when the code is ported to different virtual machines. This is done by separating high-level code analysis from low-level (JVM dependent) timing analysis. The contribution of this paper is showing how a compact high-level representation of the WCET can be delivered with the code and the final WCET of the code resolved at run-time by a simple procedure that combines these two sources of information.

The first step consists of introducing Java Class-File (JCF) attributes that support platform-independent program-path analysis (e.g. information about maximum loop bounds) so that the Worst-Case Execution Frequency (WCEF) of individual Java Byte Codes (JBC) can be derived. The second step is the analysis of how the JBCs execute on the platform/VM that is independent of the software being analysed for its WCET. Finally, the two previous stages are integrated either off-line or at run-time. The run-time option necessitates VM extensions to support the final stage of the WCET analysis only and performing of run-time checks on the WCET path information. The high-level analysis is performed at the JBC level, and no assumption is initially made on the programming language used to generate such code, neither availability of the source code is required. We have previously shown that control flow and data flow information can be extracted from the JCF and therefore source language independent WCET analysis can be performed [4].

Current techniques for WCET analysis can not be directly applied to this portable framework. This is mainly because the high-level analysis stage does not have available the execution time of individual code segments. For this reason the high-level analysis uses the notion of WCEF vectors as the data structure that holds the high-level WCET analysis.

The rest of the paper is structured as follows. In the remainder of this section we present a review of the related contributions on the topic of WCET analysis and the description of the timing features of the two Java specifications for real-time. Section 2 provides an overall view of our framework. The subsequent sections describe the components of the framework in detail: section 3 describes the high-level part of the analysis, section 4 presents the low-level part, and section 5 describes how the platform-independent and software-independent analyses are integrated. Section 6 presents the results of the evaluation. The conclusions are presented in section 7.

1.1 Worst case execution time analysis

There exists a well founded ground of research on platform-dependent WCET, for example [5]. Several approaches exist to determine the WCET of a section of object code. The basic and common approach is to build the graph of basic blocks (BB)[1] from the source code and the generated object code. The timing of each BB is determined, for instance, by adding up the WCET of each of the machine instructions within the BB. Loops are identified and annotated with the maximum number of iterations.

The timing information of each basic block can be then combined to determine the WCET of a whole program by using a timing schema [5] which is a set of rules to collapse the control flow graph annotated with timing information. Let $WCET(S)$ denote the WCET of a code segment S, and assume that we have initially the WCET of all basic blocks. From this, the WCET of a whole section of code can be determined by collapsing the graph of basic blocks applying, basically, the following rules:

- $WCET(S1;\ S2) := WCET(S1) + WCET(S2)$
- $WCET(if\ E\ then\ S1;\ else\ S2) := WCET(E) + max(WCET(S1), WCET(S2))$

  where E is the conditional code.
- $WCET(for\ \ (E)\ \ S;) \ := \ (n + 1)WCET(E) + nWCET(S)$

  where E is the loop expression and n is the maximum number of iterations of the loop. This schema can also be applied to any type of loop with bounded number of iterations.

The effects of low level features like cache effects and pipeline effects can be incorporated in the analysis at the basic block level or by determining their impact across several program paths. For example, this is usually modelled as a gain factor (negative time) in the analysis.

User annotations are usually included in the code to drive the analysis process. Simple annotations allow to define maximum loop bounds that could not otherwise be determined automatically or more sophisticated information like mutually exclusive paths [6] or dependencies between loop induction variables [3]. These annotations are usually included in the source code as specially formatted comments and they are extracted by the WCET analysis tool.

Portable WCET addresses the issue of providing a set of techniques which are widely applicable. This is achieved by providing language independent high-level analysis and platform independent analysis. High-level independence is achieved by analysing an intermediate representation rich enough to capture control flow and data flow information. Platform independent analysis is achieved by parametersising the different targets. There is some additional pessimism in performing the WCET process in this particular

[1] A BB is a continuous section of code in the sense that control flow only goes in at the first instruction and leaves through the last one

way which compensate for the added benefits that portability brings. The work presented in this paper is an investigation of a portable WCET framework based on the Java Platform and is an extension of the contributions presented in [4] and [7].

There are currently two projects for defining the requirements of real-time extensions of the Java Platform. The Real-time core extensions for the Java Platform (or Core for short) by the J-consortium [2] and the Real-Time Specification for Java by the Real-time for Java expert group [1]. Our proposed framework for portable WCET complements both approaches. We provide a mechanism for determining at run-time the WCET of a section of code by applying the timing information of the underlying virtual machine. This could be used for schedulability analysis if required. Also note that we provide a framework for computing WCET of arbitrary sections of code.

2 Framework overview

The described framework, shown in Figure 1, has been developed with three main goals in mind: WCET-analyzability, providing a means to check correct timing behavior and portability of WCET information. The following sub-sections deal with these goals.

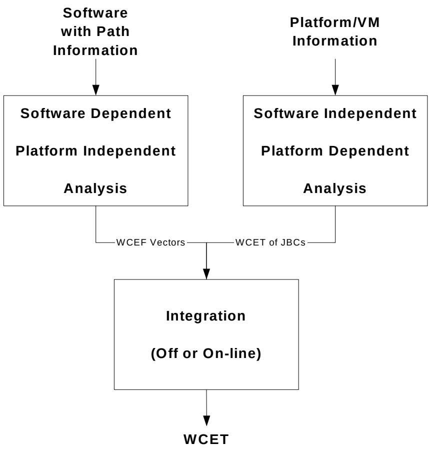

Figure 1. Portable WCET framework

To perform WCET analysis information is needed about the maximum number of executions of all statements, or instructions of the analyzed code, in the given application. Determining this maximum execution count for each code section is a problem as complex as the halting problem, i.e., there exists no automatic solution that is exact. Therefore to compute WCET bounds, the real-time programmer has to give the analysis tools hints about the maximum execution counts. To support this our framework includes a Java class[2] which allows to add user annotations for WCET. The WCETAn class and its methods are described in more detail in Section 3.

[2]An Ada version has been also defined

The WCET analysis described so far relies on the correctness of the information about execution frequencies that the programmer provided in the WCETAn-method calls. Due to the fact real-time code must under no circumstances execute longer than allocated for its worst-case execution, errors in the provided frequency information must be detected. As mentioned above, pre-run-time verification of the information is not possible. Therefore the real-time virtual machine performs runtime checks to verify the correctness of the frequency information.

Java and JBC have been designed for functional portability. To support portability the framework uses a three-step approach for the analysis. The analysis assumes that the programmer has already compiled the Java source into a class file with a Java compiler. These steps are shown graphically in figure 1.

The first step is the portable analysis. In this stage the technique analyzes JBC with calls to the methods of the WCETAn class and computes portable WCET information in the form of so-called WCEF vectors. WCEF vectors represent execution-frequency information of basic blocks and more complex code structures that have been collapsed during the first part of the analysis. The WCEF vectors returned by the first analysis step are stored back into class files as code attributes. The class files are then ready for distribution to the target VMs on which the real-time code is to run.

In parallel, analysis of the target platform is performed. This takes the form of the definition of a timing model of the VM which involves performing platform-dependent analysis (i.e. in the context of specific hardware and VM) of each JBC instruction implementation. In this stage information of potential effects of pipelines and caches may be captured.

Finally, a real-time enabled target virtual machine performs the combination of the high-level analysis with the low level VM timing model to compute the actual WCET bound of the analyzed code sections. As the resources used by this stage are manageable, it can easily be performed even on small virtual machines. The following sections present each of these steps in more detail.

3 High-level analysis

The high-level analysis takes a JCF and performs the analysis in a platform independent way. The purpose of this step is to build a compact but true representation of the timing behaviour of the code. As it is not possible to make assumptions about the timing behaviour of the individual JBC implementations we can only provide WCEF of JBC instructions.

The restrictions introduced to be able to verify the code for safe behaviour result in JBC for which control flow and data flow analysis can be performed. This allows the definition of the high-level analysis at the JBC level and therefore not requiring the source code. In addition this means that programs not written in Java but compiled for the Java platform can be also analysed for their WCET.

3.1 Predictability and sources of unpredictability

The majority of JBCs have highly predictable execution times. They tend to be fairly constant or alternatively have bounded WCETs. However, there are some exceptions that need to be addressed.

There are JBC instructions which have execution times that depend heavily on the context in which they are executed. They can be high-level object oriented instructions and multidimensional array access operations.

Assuming a WCET on these instructions will be very pessimistic as we should assume the worst possible scenario in all usages of them. The solution to 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*. It suffices to consider a linear combination, i.e. the WCET of an instruction is a function of n given by $a + bn$ where a is the execution time for the first iteration and b is the cost to perform each additional iteration. Most instructions have a fairly constant execution time and therefore have $b = 0$.

There is an additional problem to address, there are instructions which may make the code unpredictable, e.g., the invokevirtual method for calling virtual methods. In the general Java environment, this method may request another method to be retrieved from a remote host. This is clearly unacceptable in a system that needs bounded WCET. 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 it is possible to determine an upper bound on the time it takes to obtain it.

In summary, the execution time of most JBC instructions is fairly predictable, except for two cases, instructions which are data dependent and for which we analyse them with internal iteration factors and instructions which are unpredictable that are not WCET analysable.

3.2 WCET annotations

To support WCET, analysis annotations have to be provided in the source code. However, these annotations are usually introduced as specially formatted comments in the source code. This approach is not portable 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, Java and Ada versions have already been produced [4]. It allows scopes or code-block boundaries, maximum number of iterations of loops, execution modes and arbitrary path execution frequencies to be described [8].

Take the following code example. which shows how to specify that a loop iterates *50* times in the worst case, but that in the particular mode of operation called *quick_mode* it only iterates *10* times.

```
WCETAn.mode quick_mode = new WCETAn.mode();
...
WCETAn.LoopCount (50);
WCETAn.LoopCoount (10, quick_mode);
for (i=0; i<N; i++)  ...
```

The context in which this fragment is used determines the mode. This allows the analysis tool to use tighter loop bounds for different calls and therefore reduce the pessimism. Without the annotations, extra pessimism would be incurred in the analysis by considering all calls to use the maximum number of iterations.

WCET annotations are translated into the JBC as method calls. The method calls are designed in such a way that the compiler (from high-level language to JBC) does not optimise away the information and so that the information can be extracted from the code using traditional compiler analysis techniques. For example, the method calls could send data to a specific memory location. The annotation mechanism has been designed to define maximum loop bounds, to identify particular paths in the code and to ascertain certain properties of the code, for instance, mutually exclusive paths or dead paths. It also allows us to define the mode of operations and to specify the properties (i.e. loop bounds under this particular modes). The key element is the identification of *scopes* in the code and the declaration of temporal properties of the scopes. For more details on its use refer to [4]. Tool support is provided to remove all WCETAn references in the final JCF once the analysis is completed and therefore no performance penalty is incurred.

3.3 WCEF vectors

In the context of JCF/JVM, the code is compiled down to JBC and then this JBC is run on the virtual machine. This is how functional portability is achieved. However, until the VM and HW the code will run on is selected, it can not be determined how long a section of code will take to execute. In addition, since a goal for the work is portability it is useful to separate the parts of the analysis that are software dependent from those that are platform dependent.

Formally, the WCEF vector of a section of code S is denoted by $F(S) =< F_{p1}(S), F_{p2}(S), \ldots, F_{p_n}(S) >$ for $p_i \in B$. Where B is the set of Java byte codes and $F_p(S) = (a, b)$ where a is the maximum number of times that the JBC instruction, $p \in B$ is executed in the scope S and b is the maximum number of internal iterations. Note that the addition of two vectors and the multiplication of one vector by an scalar are easily defined.

The machine independent analysis also requires the identification of the maximum number of times a given method is called. In particular, methods for which there is no JBC implementation (native methods). This takes the form of a *bag* of named methods in the scope S with their associated maximum calling times. We denote it with $M(S)$ and we called them the WCEF calls. It is simply: $M(S) = \{(name_1, f_1), (name_2, f_2), ...\}$. When we refer to WCEF, we refer to both WCEF vectors and calls.

Portable WCET is achieved by determining, in the first place, the maximum execution frequency of each of the Java bytecodes in each BB and by defining a timing schema on the WCEF vectors. This allows the information to be distributed with the JCF and delay the final computation of the WCET until the details of the execution time of each of the JBC instructions is known.

3.3.1 Computing WCEF vectors

Let B be the set of JBC instructions, $F(S)$ a vector of execution frequencies of each of the instructions in B for a given code segment S and $M(S)$ the WCEF call. The WCEF vectors and calls can be built in a bottom-up fashion with an approach similar to the timing schema:

- Sequential: For two sequential segments of code S1;S2 with WCEF vectors $F(S1)$ and $F(S2)$ the WCEF of the sequence S=S1;S2 is:
  $$F(S) = F(S1) + F(S2) \qquad (1)$$
  and with WCEF calls $M(S1)$, $M(S2)$ then
  $$M(S) = M(S1) + M(S2) \qquad (2)$$
  where the addition of vectors has the usual meaning (addition of element by element). Note that the addition of pairs $(a, b) + (c, d) = (a + c, b + d)$ has the expected meaning. In the worst case the instruction is executed $a + c$ times and iterates $c + d$ times. Also the addition of WCEF call sets has the meaning of joining both sets, adding up the execution frequencies of the common call names.
- Iteration: For a loop S with header H, body B, and a maximum number of iterations n the WCEF is:
  $$F(S) = (n + 1)F(H) + nF(B) \qquad (3)$$
  and
  $$M(S) = nM(B) \qquad (4)$$
  where the addition of vectors and the multiplication of a vector by a pair (a, b) have the usual meaning. And where $nM(B) = \{(name_i, nf_i)\}$ for all $(name_i, f_i) \in M(B)$.
- Conditional: For two alternative branches $S1$ and $S2$ of a conditional statement S the WCEF of S is:
  $$F_p(S) = \max\{F_p(S1), F_p(S2)\} \;\; \forall p \in B \qquad (5)$$
  and
  $$M(S) = \max\{M(S1), M(S2)\} \qquad (6)$$
  where $\max\{(a, b), (c, d)\} = (\max\{a, c\}, \max\{b, d\})$, $\max\{M(A), M(B)\}$ is the set of names and frequencies in both A and B with the maximum frequency in either A or B. If the name only exists in one set, then it is present as well with the same frequency in $M(S)$.

To account for pipeline effects the relative frequency of pairs of instructions needs to be provided too. $F_{(p,q)}(S)$ is the WCEF of the JBC p being executed before q. As not all pairs of instructions are significant, only a subset of significant pairs need to be provided in the final WCEF. See [7] for a detailed analysis.

4 Platform-dependent analysis

The platform dependent analysis takes the form of a Virtual Machine Timing Model (VMTM) which in its simplest version is a description of the worst case execution time of each of the JBCs and additional information like WCET of native method.

Formally, let $T(\mathsf{p}) = (x, y)$ denote 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 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 the "gain time" due to pipeline effects across two consecutive JBC instructions is considered. For a sequence of JBC instructions p_1;p_2 the gain factor is written as $\lambda(\mathsf{p}_1, \mathsf{p}_2)$. The gain factor is the minimum reduction in the execution time we can expect to achieve due to the pipeline effects. For some processor architectures this gain factor can be very large. It is infeasible to provide the gain factors for all possible sequences of instructions. When calculating the gain factor, slow-down effects (such as avoiding pipeline hazards due to register contentions) also need to be accounted for. Conceivably, but unlikely, this could lead to the gain factor being positive.

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 $y = 0$). Note that as each JBC instruction is made up of several machine instructions the effects of pipelines do only affect a pair of JBCs.

Finally, the VM timing model includes the WCEF of named methods (mainly native methods). $N(name)$ is the WCET of method given by "name". Note that some methods may be unbounded and therefore the WCET can be represented as infinite.

In summary, the timing model of a VM is made up of a list of the WCET of native methods and 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. For more details can be found in [7].

5 Integrating the analysis

The final step is the determination of the WCET from the WCEF vectors and WCEF calls for a given VM timing model. This functionality should be implemented in the VM which means it has to be simple and with low resource requirements.

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

$$
\begin{aligned}
WCET_S &= \sum_{\forall p \in B} (F_p(S).a\, T(p).x + F_p(S).b\, T(p).y) - \\
&- \sum_{\forall (p,q) \in S^2} F_{(p,q)(S)}\ \lambda(p,q) \\
&+ \sum_{\forall n \in M(B).name} M(B).f\ N(n)
\end{aligned}
$$

This is simply a linear combination of the WCEF vectors, names and VMTM.

5.1 Class file attributes

The WCEF information needs to be delivered together with the JCF. This information is delivered in the JCF as an additional code attribute. A real-time enabled VM needs to provide support for reading this attribute and to make the information available to the real-time high-level analysis tool.

The portable WCEF vectors computed for methods in the first WCET-analysis step are added to the *wcef_info* tables of class files in the form of a new *Wcef* attribute. The format of *wcef_info* tables that store the WCEF information is shown in Table 1. Following the format of *attribute_info* tables for class files *attribute_name_index* is the index of the attribute name *Wcet* in the constant pool and *attribute_length* is the length of the *wcet_info* table minus the initial six bytes. The subsequent 256*2 *wcef* fields hold the execution frequencies and internal iterations of the byte codes in the method's code section, where the i-th *wcef* entry holds the WCEF of byte code i. Although not al JBC are used, the index version results in a more compact table. Then, the pairs of JBC are defined. This is a triple made of the index of the two JBCs and its execution frequency. As we would not add all possible pairs, the description of named pairs is more efficient in this case. Next, *call_table_length* gives the length of a table that lists the methods called from the current method, see Table 3. The *attribute_name_index* of a *call_info* table points to the name of a method invoked from the current method and *wcef* stores the worst-case number of invocations of this method from the current method.

Table 1. Format of a *wcef_info* Table

Type	*Name*	*Count*
u2	attribute_name_index	1
u4	attribute_length	1
u4	wcef	512
u2	pair_table_length	1
pair_info	pair_table	pair_table_length
u2	call_table_length	1
call_info	call_table	call_table_length

Table 2. Format of a *pair_info* Table

Type	*Name*	*Count*
u2	JBCindex1	1
u2	JBCindex2	1
u4	wcef	1

Table 3. Format of a *call_info* Table

Type	*Name*	*Count*
u2	attribute_name_index	1
u4	wcef	1

5.2 Support for WCET analysis in the JVM

As mentioned previously, each target virtual machine of our framework stores its own virtual machine timing table. This timing table characterizes how long each byte code maximally takes to execute on the machine. In addition to this timing table the VM uses a *wcef_info* table to store the WCETs it has computed for the methods of the real-time application. Given the VM timing table and the method timing table, the WCET-computation step performed on the target is relatively simple. To obtain the WCET of a method the VM first reads the method's *wcet_info* table and computes the WCETs for all byte codes. Second, the VM adds the WCETs of all calls to other methods to the WCET computed so far, i.e., for each method in the *call_info* table it searches the method-WCET table for the method's WCET. If the WCET of the method has already been stored in the table, then the WCET of the method is multiplied by the number of its calls and the result is added to the WCET value computed so far. If the WCET of the method is not yet available, then its WCET is computed by a recursive call of the WCET analyzer and stored in the method-WCET table. After that recursive call the analysis for the current method continues.

This support is implemented as an special WCET class that can be incorporated as a package by the VM. The class essentially provides a mechanism for:

- Identification of whether the VM has portable WCET support.

- Interrogation of the current WCET code attributes available for a particular class and a list of the scopes for which WCEF info has been computed. One would expect to have one WCET scope defined for each instance of a real-time schedulable entity.

- Computing the WCET of a named scope.

This functionality would then be used, for instance by the scheduler (or schedulability analysis component) to determine the schedulability of the system.

6 Evaluation

In this section we present the complete analysis of a code example. The code fragment is a sort routine written in Java and compiled with the Sun `javac` compiler. We present the analysis for two virtual machine timing models described in section 4.

6.1 High-level analysis

The code is shown in figure 2. To make it clearer we assume simplistic (pessimistic) loop bounds. An accurate analysis would contemplate that the inner loop body can only be executed $\frac{10*9}{2}$ times in the scope of the function (not the 9^2 times the simplistic analysis considers). The code generated by the `javac` compiler is actually made up of 14 different Java Byte Codes only. The graph of basic blocks is shown in figure 2. The two nested loops can be easily identified: B6, B3, B4 and B5 and B8, B3, B4, B5, B2 and B7. Note that the compiler generates the head of the loop down in the code. The WCEF of each basic block and the final WCEF obtained by applying the timing schema described in section 3.3 is shown in table 4. The evaluation of the example results in:

$$B1 + 10B8 + 9(B2 + 10B6 + 9(B3 + B4 + B5) + B7) + B9$$

Table 4. WCEF vectors of the Basic Blocks and of the collapsed BubbleSort function

	B1	B2	B3	B4	B5	B6	B7	B8	BSort
ALOAD_0	0	0	2	4	0	0	0	0	486
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	405
IASTORE	0	0	0	2	0	0	0	0	162
ICONST_0	0	0	1	0	0	0	0	0	81
ICONST_1	0	1	0	2	0	0	0	0	171
IF_ICMPLE	0	0	1	0	0	1	0	0	163
IFGT	0	0	0	0	0	0	0	1	10
IINC	0	0	0	0	1	0	1	0	90
ILOAD_1	0	0	0	0	0	1	0	1	92
ISTORE_1	1	0	0	0	0	0	0	0	1
ISUB	0	0	1	2	0	0	0	0	243

```
public static void BubbleSort(int a[])
{ int i, j, t;
  int size = 10;
  WCETAn.LoopCount(9);
  for (i=size-1; i>=1; i--)
  { WCETAn.LoopCount(9);
     for (j=1; j<=i; j++)
     { if (a[j-1] > a[j])
       {t      = a[j-1];
        a[j-1] = a[j];
        a[j]   = t;
} }  } }
```

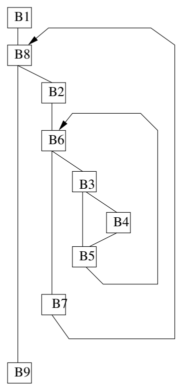

Figure 2. BubbleSort and Basic Blocks Graph

6.2 VMTM examples

We have investigated the VM timing model of two architecturally different virtual machines: Kaffe and Komodo. The same analysis could be applied to other VM's.

Kaffe [3] is an open source VM that can run on different platforms. It provides an interpreter and a JIT compiler and although it has not been designed for real-time, it has temporal properties that make it adequate for the purpose of our analysis. In order to determine the execution times of this VM we have run the Caffeine benchmarks [4] and obtained cycle accurate measurements of the execution time of each of the JBC instructions. These timings include both the overheads associated with the VM and the gain time of the pipeline effects. Therefore we assume $\lambda(p, q) = 0$. The results are shown in table 5. It shows the WCET in cycles of some of the JBC's.

Komodo is a stand alone Java on Chip VM [9] which has some nice properties. Its main feature is that it has a zero cost context switch time. The other main feature is that it has a 5 stage pipeline, and all but a very few have a gain factor of 5. Thus, a potential execution of one instruction in one cycle is possible. In this case almost all pairs of

[3] http://www.kaffe.org
[4] http://www.caffeine.org

instructions have a $\lambda(p, q) = 5$ and for some instructions including control flow ones the gain factor is reduced to 3.

Table 5. VM timing models of the Kaffe and Komodo virtual machines

	Kaffe	Komodo
ALOAD_0	67	6
BIPUSH	68	6
GO_TO	63	9
IALOAD	67	14
IASTORE	65	15
ICONST_0	60	6
ICONST_1	60	6
IF_ICMPLE	77	9
IFGT	79	9
IINC	61	6
ILOAD_1	76	6
ISTORE_1	68	6
ISUB	69	6

6.3 Low-Level Analysis

The description of the VMTM has been presented in section 4. As the two processor architectures are completely different as well as their clock cycles, the resulting figures for both tables are essentially different.

The final part of the analysis is the integration of the two data structures to obtain the final WCET. For the Kaffe VM, this results in 128703 cycles (257μs at 500MHz), for the Komodo VM this corresponds to 16737 cycles (523μs at 32MHz). Note that we are not comparing which VM is faster, we are just showing how the same analysis can produced the final WCET in a very simple way.

7 Conclusion

We have presented a framework for performing portable WCET in JVM architecture that it is based on the separation of the WCET analysis in two parts, a machine independent part and a machine dependent part. The machine independent part is performed off-line and builds the WCEF vectors and WCEF names. That is, the worst case execution frequency of the JBCs and native method calls. This information is stored back into the JCF as a code attribute. In order for the code to be analysable it has to include annotations on the worst case behaviour of its constructs (i.e. maximum loop bounds). This is done by means of extracting the information introduced by calls to the WCETAn class directly from the JBC.

Similarly the platform specific details are analysed off-line to derive a VM timing model. The VM timing model consists of definitions of how long each JBC takes to execute, in the worst case, on the given platform, the gain factor that can be expected from particular pairs of instructions and WCET of VM native method implementations.

A WCET enabled VM implementation is then able to extract this WCEF information and combine it with the VM timing model, which is a description of the WCET of each JBC and native method. The final calculation of the WCET is therefore very simple and with low overheads and memory cost and could be incorporated into current proposals for Java and real-time.

A case study has been presented that shows how a bubble sort can be performed on two VMs (Kaffee VM and Komodo VM) at run-time.*

References

[1] G. Bollella, J. Gosling, B. Brosgol et al. *The Real-time Specification for Java*. Addisson Wesley, 2000. http:\\www.rtj.org.

[2] International J Consortium Specification, "Real-Time Core Extensions for the Java Platform." Specification No. T1-00-01. http://www.j-consortium.org, 2000.

[3] G. Bernat and A. Burns, "An approach to symbolic worst-case execution time analysis," in *25th IFAC Workshop on Real-Time Programming. Palma (Spain).*, May 2000.

[4] 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*, pp. 81–88, June 2000.

[5] A. Shaw, "Reasoning about time in high-level language software," *IEEE Transactions on Software Engineering*, vol. 17, no. 7, pp. 875–889, 1989.

[6] R. Chapman, A. Burns, and A. Wellings, "Integrated program proof and worst-case timing analysis of Spark Ada," in *proc. ACM Workshop on language, compiler and tool support for real-time systems*, ACM Press, 1994.

[7] I. Bate, G. Bernat, G. Murphy, and P. Puschner, "Low-level analysis of a portable wcet analysis framework," in *6th IEEE Real-Time Computing Systems and Applications (RTCSA2000). Cheju Island, South Korea*, pp. 39–48, December 2000.

[8] P. Puschner and A. Schedl, "Computing maximum task execution times - a graph based approach," *Real-Time Systems*, vol. 13, pp. 67–91, July 1997.

[9] J. Kreuzinger, A. Schulz, M. Pfeffer, and Th.Ungerer, "Real-time scheduling on multithreaded processors," in *7th international RTCSA. Cheju Island, South Korea*, December 2000.

[0]This work has been partially funded by the European Union under project NextTTA IST-2001-32111