Portable Worst-Case Execution Time Analysis Using Java Byte Code

Guillem Bernat, Alan Burns and Andy Wellings *
Real-Time Systems Research Group
Department of Computer Science
University of York
YO1 5DD - York (UK)
{bernat,burns,andy}@cs.york.ac.uk

Abstract

In this paper we address the problem of performing worst case execution time (WCET) analysis of Java Byte Code (JBC) which may be generated from different compilers and from different source languages. The motivation for the framework presented is to provide WCET analysis which is portable and therefore more likely to be used in an industrial context. Two issues are addressed in this paper, how to extract data flow and control flow information from JBC programs and how to provide a compiler/language independent mechanism to introduce WCET annotations in the source code. We show that an annotation mechanism based on calls to a static class with empty methods result in similar code when generated by Java or Ada compilers.

1 Introduction

Of seminal importance to virtually all forms of scheduling analysis is the production of a worst-case execution time (WCET) profile for the tasks/processes/threads that constitute the application's code. Once the WCET values are known then any particular scheduling scheme chosen for use can be parameterised and predictions of the overall temporal behaviour of the system can be established. The process of acquiring estimates of WCET is known as *timing analysis*.

Since the first papers on timing analysis appeared in 1986 [6] there has been an impressive procession of techniques and algorithms (a review of this material has recently been produced, [11]). By contrast the industrial take up of timing analysis has been comparatively poor. Industrial practice is still focused on measurement, with its inherent drawback of optimism - i.e. WCET values obtained may not indeed be the worst-case. The reasons that timing analysis is not more systematically used can be accounted for by the lack of available tools. And the poor provision of tools is due to the need to closely associate any particular form of analysis with:

- a single source language
- a specific compiler
- a particular version of the compiler and compiler switches
- a unique board layout with processor spec, memory configuration, clock speed etc

Lack of portability therefore hinders the take-up of, what are now, well established techniques. In this paper we attempt to improve this situation by proposing a timing analysis scheme based on Java Byte Code (JBC). This form of intermediate representation is now becoming a defacto standard. Its existence facilitates the production and use of a range of static analysis techniques focused on the JBC. Note, we are not concerned just with Java. Many language compilers now target JBC (including Ada and C translators). Moreover we are not concerned with JBC interpreters. We assume JBC is 'compiled' down to a native assembler/machine code. The use of JBC presents the following challenges:

- How to pass necessary semantic information required to perform WCET analysis from the source to the JBC - and to do so in a compiler independent way.
- How to analyse and manipulate the JBC in order to undertake WCET analysis.
- How to incorporate knowledge about the target hardware into the analysis.

We address, in this paper, the first two challenges. In Section 2 we present an introduction to Java Byte Code and

*This work has been funded by the DERA with project CSM/1254.

the Java Virtual Machine. In Section 3 we describe the requirements for the WCET analysis. Finally, in section 4 we present the annotation mechanism for high level analysis of JBC.

1.1 Worst-case execution time

WCET techniques are usually classified in two categories, low level and high level. Low level techniques address the timing issues of machine code, like for example, the effects of cache and pipeline. High level analysis are more concerned with structural aspects of the program.

It is generally not possible to obtain the WCET by analysing the source code only. For example, it may not be possible to determine an upper bound on the number of iterations a loop will perform even though they are bounded, for this reason annotations are included in the code to express such bounds. For a discussion of general architecture of WCET tools see [3].

We follow the same style of annotations of the Spark Ada [1] in which annotations are introduced, mainly, for program analysis and verification. Chapman in [2] extends these annotations to perform WCET of Spark Ada. These annotations result in much tighter WCET bounds. This was achieved by expressing function/procedure modes and mutually exclusive paths. We introduce a compiler-transparent annotation mechanism that captures the same type of information Spark Ada does.

2 Java Byte Code

Java Byte Code (JBC) is the object code of a stack-based virtual machine. A program is compiled into a *Java Class File* which holds the definitions of constants, methods and other information as well as the JBC. The Java Virtual machine is the name of an engine that actually executes a JBC program. The basic model of a virtual machine is a piece of software that loads and interprets the instructions of a JBC file and performs the operation that the instruction represents. The portability is achieved by porting, only, the virtual machine for each different platform (with some extra classes like I/O, network, GUI, ...). For a detailed description of JBC and Java class files, see [7].

2.1 Java Virtual Machine

Instead on relying on registers and address modes, the Java virtual machine (or the instructions of JBC) is stack-based and it has simple address modes. All operations are performed through the operand stack. This allows an efficient implementation of the virtual machine in processors which have few registers and few addressing modes. More sophisticated processors can also perform some mapping of their advanced features into these instructions.

There are three main types of implementations of the virtual machine

- Interpreted: the JVM is an executive that interprets and executes each JBC instruction.
- Just in time (JIT) compiler: the first time a method is invoked, it is compiled into object code, subsequent calls benefit from code already in object format.
- AOT (ahead of time) compiler: All code is translated to object code before execution. In this case, the VM is in fact a compiler rather than an "execution" machine.

JIT provides good average performance but it is not suitable for real-time systems as the worst-case computation time has to include the time to compile the code, which is very pessimistic, and very difficult to predict.

Both the interpreted model and the AOT are predictable from the timing perspective. However, the code generated by the interpreted model has a low performance. The AOT is the one which shows the maximum performance and predictability, and therefore, suitable for real-time systems.

The issues of memory management, and more specifically, garbage collection should be also addressed. For the purpose of our application context we assume that a real-time memory model (i.e. real-time garbage collection) is used.

2.2 The Java class verifier

The security issues that drove the development of Java have resulted in a Java class file structure that can be analysed beforehand for any malicious or suspicious behaviour. This is done by the Java class verifier.

The Java class file verifier scans the Java class file and performs some checks that guarantee that the code is correct (from very simple checks like that the correct number of constants have been declared, to testing the proper use of the stack, like controlling that all parameters pushed on the stack in a call are popped back correctly). The verifier is usually applied to a class file when it is loaded by the Java Virtual Machine. The a-priori check on the JBC allows very efficient implementations of the Java virtual machine because it does not need to perform run-time checks apart from some explicit cases like the usage of reference casts with the `checkcast` instruction.

One consequence of particular importance for WCET analysis is that a verified code has the following property: *for any possible path from the start of the program to any instruction, the stack of each of the paths is in a compatible state*. Here, *compatible state* means that the stack for each of the possible paths has the same number of elements, and that the types at each position of the stack are identical in each of the stacks. In fact, it is a little bit more elaborate because it also checks that reference types contain a reference to the first common superclass of the two types.

This means that the stack depth can be determined off-line (and therefore no stack overflow/underflow can occur). Thus, it is possible to determine off-line, at which depth each instruction that pushes/pops information on/from the stack will store/load it. This is very important for data flow analysis. We later show that in order to perform data flow analysis the JBC has to be translated into a three address code representation. In this new representation, the stack is represented as a set of variables, and all stack operations are translated as memory operations with these variables. This mapping could not have been possible if the actual stack position used by a particular instruction were not known.

2.3 Real-Time Java

There are currently two projects for defining the requirements of real-time extensions of the Java Platform. The J consortium Specification [5] and the Real-Time Specification for Java [13]. Only the first one, which is based on the PERC system [9] clearly addresses the issue of requiring time analysable code. However, they impose strong requirements in the code (maximum loop bounds detected off-line) which are too restrictive and their extensions only address the Java language.

3 Portable WCET analysis

In order to perform WCET of JBC, some information on the structure of the code is required. This information, like loop constructs and induction variables, is generally available at source code level. Traditional WCET tools rely on the interaction between the compiler and the WCET tool, or on the availability of the source code. However, to achieve portable WCET we require not to rely on information at the source code, neither on the support of the compiler to provide such information. We need to be able to extract the information from the Java class file alone.

This problem is very similar to trying to perform high level analysis of low level code, although, there are significant differences between JBC and other machine level codes. The main one is due to the fact that JBC is verified. This imposes a structure on how the code is generated and the constructs used. As we show in the following section, both control flow and data flow information can be extracted from JBC.

3.1 Predictability

The main requirement to perform WCET is predictability. This is required at several levels.

- Source program level: It is assumed that source programs are themselves predictable and that WCET can be determined. For instance, no unbounded loops are used, etc.

- Library level: Code in standard libraries used by the programs should be also predictable, or at least should have bounded worst-case execution times.

- Compiler level: The code generated by the compiler should be also predictable.

  These three first requirements are common requirements for any type of WCET analysis.

- JBC instruction set. Each JBC instruction has to be predictable itself. Most JBC instructions have very predictable execution time, for instance `iadd` gets two elements from the stack, adds them together and pushes the result on the stack. There are several JBC instructions which are quite complex and and their time behaviour depends on the context in which they are called, for instance: `Throw, invokespecial, checkcast, instanceof`. Although, the behaviour can vary considerably between different invocations (depending on the context in which they are called), their behaviour is also bounded.

- Virtual machine implementation: The WCET is greatly affected by the way the virtual machine is implemented. The code generated by the VM has to be also predictable. This includes, memory management, garbage collection, object access, etc. Delays due to the interaction with other tasks due to using synchronized methods belong to scheduling analysis (rather than timing analysis), although the cost of handling the internal queues has to be accounted for in the timing model of the virtual machine.

3.2 Virtual Machine Timing model

The timing behaviour of a virtual machine depends on how it is implemented and on the hardware it runs on. The particular WCET of a piece of JBC will depend on the time each JBC instruction takes to execute. To support portable WCET we need to isolate these two aspects. For this reason a timing model of the virtual machine for each particular target the code may run on is required.

A timing model is a description of how long does each JBC instruction take to execute. A simple timing model is to consider the maximum execution time of each JBC instruction. Computing the WCET of the whole program is therefore determined by combining high level analysis with the timing model of the VM. The advantage of this approach is that it is very simple and easy to perform. However, it may be unnecessarily pessimistic. The refinement of this model is to include the actual code generated by the AOT for each usage of the JBC instructions. This is more accurate as the specific optimisations performed, depending on the context in which an instruction appears, are considered. With this approach, tight low level analysis which include cache and pipeline effects can be also included. For hardware implementations of the VM [10] a similar approach is devised.

The focus of this paper is on high-level analysis of JBC and transparent annotation mechanisms, the issue of defining timing models of the virtual machine is still under research. For the purpose of this paper we assume a simple virtual machine timing model.

4 High level analysis of JBC

In this section we show the how high-level analysis can be performed from the JBC. The result is comparable to the information that could be extracted from the analysis of the source code. The constrained structure of JBC enables full data flow and control flow analysis to be performed. In this context, it is shown how annotations embedded in the JBC can be identified and extracted. Although JBC is considered a low-level code, it has a very rigid structure that allows accurate analysis. As opposed to other low level machine code formats where it is hard (or even impossible) to perform this type of analysis.

4.1 Control Flow

The JBC instructions that can affect the flow of control of the program are very well defined. They are: unconditional control flow, control flow, case, method invocation, and exception. There are no mechanisms to change the control flow of a JBC program by other means than by using these instructions. For instance, loading an address into a variable and then jumping to that address is forbidden. All jumps are performed to known addresses. Thus determining the boundaries of basic blocks of the code (sequences of instructions for which control flow can only occur at the first instruction or at the last instruction), and the graph of basic blocks is straightforward.

4.2 Data Flow

Data flow information and data dependencies are difficult to extract directly from JBC because of the stack base representation. For this reason a translation of JBC to a generic three address code format is needed. This translation is also performed on some efficient JIT or AOT compilers to optimise the number of memory operations [4].

The intrinsic restrictions JBC has, originally introduced for security purposes, makes JBC analysable. Using traditional compiler techniques [8], loops can be detected, induction variables extracted and all sorts of data dependencies established, in a very similar way as they could be extracted from high level languages.

One of the principal problems in WCET is determining loops, and bounding the number of times each loop can execute. Several techniques exist for such purpose [12]. Two approaches are possible. To automatically extract and detect loops bounds and/or rely on information provided by the programmer to bound the number of loop iterations. In any case, there are some loops for which no automatic tool can extract maximum execution times and therefore annotations are required.

As shown in section 4.1 the program flow can be extracted. From this structure standard compiler techniques can be used for identifying loops, extracting loop headers, and exit points [8]. Using the result from data flow analysis, basic and derived induction variables and the range of increment can be also extracted. Also, the expressions that determine the exit conditions of the loop can be identified. By performing loop extraction at the JBC level, different loop constructs generated differently from different compilers are easily dealt with. If loop bounds are static and expressed as constants in the code, they can be identified and extracted straight away. More complex structures need the support for annotations which is the subject of the following section.

4.3 Annotations

Our approach is to introduce annotations in the source code as calls to a predefined class (the WCETAn class) which are dummy. Compilers would have to generate the calls into the JBC which are then identified, extracted and analysed by the WCET tool. Furthermore, all code associated to these calls can even be removed from the Java class file as they do not have any impact in the functional aspects of the code. The only special requirement we need is to guarantee that the compiler would not eliminate this calls, neither will move them in the code. This is a functionality commonly available in compilers.

The rest of this section examines the structure and usage of this class file for both Ada and Java programs.

The annotation mechanism is based on a class, the WCETAn class, that consists of a set of procedure calls that encapsulate the annotations of the code. Following several annotation mechanisms [2], the type of information that is required to annotate is:

- Code block boundaries.
- Types of loops: infinite loops, finite loops.
- Maximum number of iterations of loops
- Maximum number of iterations of nested loops

- Bounds on execution times of subprogram calls.
- Moded versions of the above.

The only requirement we need from the compiler is to guarantee that the compiler preserves the name of the procedure in the JBC so that the WCET tool can identify it, and that the position of the call in the object code is not moved.

In general, we need a mechanism for:

- Creating Tags: entities (i.e. code blocks) are identified with tags, therefore a mechanism to create unique identifiers is required,
- Naming blocks of code: particular tags are identified with blocks of code.
- Asserting properties: Name loop bounds, identify dead paths, moded function calls, beginning and ending of the sections of code to be analysed, etc.

We also require a mechanism that will generate code which is traceable, and that the code generate by different compilers will be similar (and also with compiler of other languages). This is achieved by declaring static classes and methods. Tags correspond to (empty) static classes. Code assertions are static methods. The annotation class is as follows:

```
public class WCETAn {
   // WCET tags
   static class Mode  {}; // method modes
   static class Label {}; // section of code

   // Naming tags
   static void Define_Mode   (Mode  m) {};
   static void Use_Mode      (Mode  m) {};
   static void Identify_Code (Label l) {};

   // Assertions
   static void Loopcount  (int   n          ) {};
   static void Loopcount  (int   n, Mode  m) {};
   static void Dead_Path  (Mode  m, Label l) {};
   static void Begin_WCET (Label l)          {};
   static void End_WCET   (Label l)          {};
   ...
};
```

A compiler will either translate static classes as static variables or static fields. All method invocations are calls to invokestatic. There is also code generated for initialising the new classes included, which can be ignored.

For example, consider the following piece of code which is a fragment of a pow function [2] which computes the n^{th} power of a floating number with a loop. For this particular piece of code, we want to express the fact that the maximum exponent that we can encounter is 10, therefore the maximum number of loop iterations is bounded by 10. But we also want to consider a particular mode of the pow function (denoted by power_A) in which, the maximum loop bound is only 2. We also want to mark that the if construct evaluates always to false under this particular mode thus we need to mark the then part of the if as dead in mode power_A. This is depicted in the following code:

```
public  class use_pow {
 // declare modes for function pow
  static WCETAn.Mode Power_A = new WCETAn.Mode();

  static float pow (...)
  {  // label for then block
     WCETAn.Label Then1 = new WCETAn.Label();

     // tag Power_A  as a mode of Pow
     WCETAn.Define_Mode(Power_A);

     if (Some_condition)
     {     WCETAn.Identify_Code(Then1);
           Some_Code;
     }
     else
     {     Some_Other_Code;
     }
     // Loop iterations depend on the mode.
     WCETAn.Loopcount(10);
     WCETAn.Loopcount(2,Power_A);
     for (i=1; i<= some_number; i++)
     { loop_body; }
     // in mode A, then1 is never taken
     WCETAn.Dead_Path(Then1,Power_A);
  }
  ...
  // calls to foo can be simple or moded:
  public void foo (){
     float f,g; int i;
     static WCETAn.Label WCET_foo = new WCETAn.Label();

     WCETAn.Begin_WCET(WCET_foo);
       ...
       // this is a general call to pow
       f=pow(g,i); // use worst-case
       ...
       // this is constrained call to pow
       WCETAn.Use_Mode(use_pow.Power_A);
       g=pow(f,2); // results in tighter WCET
       ...
     WCETAn.End_WCET(WCET_foo);
}
```

This code fragment illustrates the three uses of the annotations presented in this paper. A function, declares new instances of Mode and Label classes which are associated with elements in the code by calls to Define_Mode and Identify_Code. The boundaries of the code of a Identify_Code are the outermost block in which it is declared. The code fragments to perform WCET are marked with calls to Begin_WCET and End_WCET.

The same package is also written in Ada, the empty static classes are translated as empty record types.

```
package WCETAn is
      type Mode  is private;
      type Label is private;
      procedure Define_Mode  (m : in Mode);
      procedure Use_Mode     (m : in Mode);
      procedure Identify_Code(l : in Label);
      procedure Loopcount    (n : integer);
      procedure Loopcount    (n : integer;
                              m : Mode);
      procedure Dead_Path    (l : Label;
                              m : Mode);
      procedure Begin_WCET   (l : Label);
      procedure End_WCET     (l : Label);
   private
      type Mode  is record null; end record;
      type Label is record null; end record;
```

```
      -- procedures with empty bodies
   end WCETAn;
```

The Ada version of the same code fragment of the pow function is therefore:

```
 with WCETAn;
 package use_pow is
   -- Declare modes for function pow
   Power_A  : WCETAn.Mode;

   procedure pow (...) is
      -- label for then block
      Then1 : WCETAn.Label;
   begin
      -- tag Power_A as  a mode of Pow
      WCETAn.Define_Mode(Power_A);

      if Some_Condition then
            WCETAn.Identify_Code(Then1);
            Some_Code;
      else
            Some_Other_Code;
      end if;

      -- Loop iterations depend on the mode
      WCETAn.Loopcount(10);
      WCETAn.Loopcount(2,Power_A);
      for i in 1 .. some_number loop
         Loop_Body;
      end loop;
      WCETAn.Dead_Path(Then1, Power_A);
   end pow;
   ...
   procedure foo is
      f,g      : float;
      i        : integer;
      WCET_foo : WCETAn.label;
   begin
      WCETAn.Begin_WCET(WCET_foo);
         ...
         -- general call to pow
         f := pow(g,i);
         ...
         -- constrained call to pow
         WCETAn.Use_Mode(use_pow.Power_A);
         g := pow(f,2);
         ...
      WCETAn.End_WCET(WCET_foo);
   end foo;
end use_package;
```

The resulting JBC from the Java language, and from the Ada language is very similar. For instance, consider the declaration of the Then1 label and the declaration that Then1 is dead in mode Power_A. The following is an excerpt of the disassembled code generated by the JDK 1.2.2 compiler of the creation and definition of label1:

```
   0 new #2 <Class WCETAn$Label>
   3 dup
   4 invokespecial #6 <Method WCETAn$Label.<init>()V>
   7 astore 7
  ...
  52 aload 7
  54 invokestatic #9 <Method WCETAn.
        Identify_Code(LWCETAn$Label;)V>
```

The new construct, is mapped into the new JBC of class WCETAn$Label which is stored in variable 7. The call to Identify_Code is performed on variable 7 [1]. The same JBC

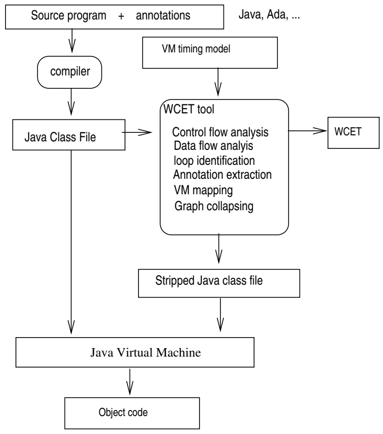

Figure 1. Structure of WCET analysis

fragment from the Ada code, as generated by the Aonix 7.0 compiler, is as follows:

```
  14 new #34 <Class wcetan.Label>
  17 dup
  18 invokespecial #36 <Method wcetan.Label.<init>()V>
  21 astore_3
  ...
  66 aload_3
  67 invokestatic #46 <Method WCETAn.
         Identify_Code(Lwcetan/Label;)V>
```

The code is almost identical, except for constant numbers. Other compilers may generate the code differently, however, a similar pattern is used. The WCET tool traces specific calls to the WCETAn class and traces back the type of arguments used, and most important the variable numbers used. Note that the actual value of the variables/classes used is not important, what is important is to be able to trace declarations and usage of the "same" object in the code. A specific test code is designed to determine the way a particular compiler generates the code for later use in the identification of the calls to the annotations class. See the annex for the full code.

4.4 Tool Support

The ideas presented in this paper are currently being implemented in a prototype tool called Javelin. The context in which the tool is used is shown in figure 1. Javelin parses Java class files, performs control flow analysis, data flow analysis and loop identification. Then, annotations are extracted and converted into annotations of the resulting context tree graph. The timing model of the virtual machine is used to time each basic block. Finally, a technique to compute the WCET of the resulting graph (i.e. graph collapsing for simple models or integer linear programming techniques for more complex graphs) is used to determine the WCET [12]. The latest status of the tool can be obtained from http://www.cs.york.ac.uk/~bernat/javelin.html

[1] More information of what is stored in a variable can be also extracted from the additional information stored in the Java class file

We are currently developing a timing model of the LORCAS Virtual Machine which has been developed at the University of York. This will complete the development of a portable code infrastructure based on JBC technology.

5 Conclusion

In this paper we have analysed the properties of JBC. We conclude that JBC is suitable for portable WCET analysis. We show that it can be performed independently of the source language and compiler used.

We have shown that control flow and data flow information can be extracted from verified JBC, support for annotations of the code is performed with calls to the WCETAn class/package. A prototype tool is being developed that analyses Java Class files and that identifies and extracts the annotations in the JBC. The mechanism is flexible enough to be extended to other types of annotations. The high level analysis of the code is complemented with the timing model of the Java Virtual Machine. Standard WCET techniques are then used to analyse the resulting context graph and to determine the actual WCET of the code.

References

[1] B. Carré, T. Jennings, F. Maclennan, P. Farrow, and J. Garnsworthy. SPARK: the SPADE Ada kernel (edition 3.1). *Program Validation Ltd.*, 1992.

[2] R. Chapman, A. Burns, and A. Wellings. Combining static worst-case execution time analysis and program proof. *Real-Time Systems*, 11:145–171, 1996.

[3] J. Engblom, A. Ermedahl, M. Sjodin, J. Gustafsson, and H. Hansson. Towards industry strength worst-case execution time analysis. Technical Report DoCS 109, DoCS Uppsala University, 1999.

[4] C. A. Hsieh, M. T. Conte, T. L. Johnson, J. Gyllenhaal, and W. Hwu. Using NET to capture performance in Java-based software. In *IEEE Computer*, pages 67–75, 1997.

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

[6] E. Kligerman and A. Stoyenko. Real-Time Euclid: A language for reliable real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9):941–949, 1986.

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

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

[9] Newmonics Inc. Perc 2.01, 1999.

[10] O'Connor, J. Michael, and M. Tremblay. PicoJava-I: The Java virtual machine in hardware. *IEEE Micro*, 17(2):45–53, 1997.

[11] P. Puschner and A. Burns. A review of WCET analysis. guest editors introduction. *Real-Time Systems (To appear)*, 2000.

[12] P. Puschner and A. B. Schedl. Computing maximum task execution times - a graph based approach. *Real-Time Systems*, 1(27), 1991.

[13] The Real Time for Java Experts Group. Real-Time Specification for Java v: 0.9.1. http://www.rtj.org, Feb, 2000.

Appendix: full POW example

This is the full example of the POW function with two modes, Power_A when exponent is equal to 2 and and Power_B when exponent is equal to 3, and the corresponding dead path annotations.

```
with WCETAn;
package use_pow is
 Power_A, Power_B  : WCETAn.mode;
 function POW (Base: in float; Exponent: in integer)
              return float;
end use_pow;

package body use_pow is
 function POW (Base: in float; Exponent: in integer)
              return float is
    Exchange    : boolean;
    aux, result : float;
    Then1, Then2, Else1, Else2 : WCETAn.label;
 begin
    WCETAn.Define_Mode(Power_A);
    WCETAn.Define_Mode(Power_B);
    if Exponent >= 0 then
       WCETAn.Identify_Code(Then1);
       Exchange:=False;
    else
       WCETAn.Identify_Code(Else1);
       Exchange:=True;
    end if;
    aux := 1.0;
    WCETAn.Loopcount(10);
    WCETAn.Loopcount(2,Power_A);
    WCETAn.Loopcount(3,Power_B);
    for i in 1 .. Exponent loop
       aux := aux * base;
    end loop;
    if Exchange then
       WCETAn.Identify_Code(Then2);
       result := 1.0 / aux;
    else
       WCETAn.Identify_Code(Else2);
       result := aux;
    end if;
    WCETAn.Dead_Path(Else1, Power_A);
    WCETAn.Dead_Path(Then2, Power_A);
    WCETAn.Dead_Path(Else1, Power_B);
    WCETAn.Dead_Path(Then2, Power_B);
    return result;
 end  POW;
end use_pow;
```

And the disassembled code is as follows. Note the following: line numbers indicate the offset in the JBC as opposed to the instruction number; there is a static class that initialises the Power_A and Power_B modes; boundaries of basic blocks are indicated with dashed lines; basic blocks 113:120 and 123:129 form a loop.

```
Method float POW(float, int)
---------------------------------------------------
   0 getstatic #28 <Field use_pow_.POW__elab_bool Z>
   3 ifne 14
---------------------------------------------------
   6 new #30 <Class java.lang.ClassCircularityError>
```

```
  9 dup
 10 invokespecial #32 <Method java.lang.
       ClassCircularityError.<init>()V>
 13 athrow
---------------------------------------------------
 14 new #34 <Class wcetan.Label>
 17 dup
 18 invokespecial #36 <Method wcetan.Label.<init>()V>
 21 astore 5
 23 new #34 <Class wcetan.Label>
 26 dup
 27 invokespecial #36 <Method wcetan.Label.<init>()V>
 30 astore 6
 32 new #34 <Class wcetan.Label>
 35 dup
 36 invokespecial #36 <Method wcetan.Label.<init>()V>
 39 astore 7
 41 new #34 <Class wcetan.Label>
 44 dup
 45 invokespecial #36 <Method wcetan.Label.<init>()V>
 48 astore 8
 50 getstatic #17 <Field use_pow.Power_A Lwcetan/Mode;>
 53 invokestatic #42 <Method WCETAn.
      Define_Mode(Lwcetan/Mode;)V>
 56 getstatic #19 <Field use_pow.
      Power_B Lwcetan/Mode;>
 59 invokestatic #42 <Method WCETAn.
      Define_Mode(Lwcetan/Mode;)V>
 62 iload_1
 63 iconst_0
 64 if_icmplt 77
---------------------------------------------------
 67 aload 5
 69 invokestatic #46 <Method WCETAn.
       Identify_Code(Lwcetan/Label;)V>
 72 iconst_0
 73 istore_2
 74 goto 84
---------------------------------------------------
 77 aload 7
 79 invokestatic #46 <Method WCETAn.
       Identify_Code(Lwcetan/Label;)V>
 82 iconst_1
 83 istore_2
---------------------------------------------------
 84 fconst_1
 85 fstore_3
 86 bipush 10
 88 invokestatic #50 <Method WCETAn.Loopcount(I)V>
 91 iconst_2
 92 getstatic #17 <Field use_pow.Power_A Lwcetan/Mode;>
 95 invokestatic #53 <Method WCETAn.
          Loopcount(ILwcetan/Mode;)V>
 98 iconst_3
 99 getstatic #19 <Field use_pow.Power_B Lwcetan/Mode;>
102 invokestatic #53 <Method WCETAn.
          Loopcount(ILwcetan/Mode;)V>
105 iload_1
106 iconst_0
107 if_icmple 132
---------------------------------------------------
110 iconst_1
111 istore 9
---------------------------------------------------
113 fload_3
114 fload_0
115 fmul
116 fstore_3
117 iload 9
119 iload_1
120 if_icmpeq 132
---------------------------------------------------
123 iload 9
125 iconst_1
126 iadd
127 istore 9
129 goto 113
---------------------------------------------------
132 iload_2
133 ifeq 163
---------------------------------------------------
136 aload 6
138 invokestatic #46 <Method WCETAn.
       Identify_Code(Lwcetan/Label;)V>
141 fconst_1
142 fload_3
143 dup
144 fconst_0
145 fcmpl
146 ifne 157
---------------------------------------------------
149 new #55 <Class java.lang.ArithmeticException>
152 dup
153 invokespecial #57 <Method java.lang.
       ArithmeticException.<init>()V>
156 athrow
---------------------------------------------------
157 fdiv
158 fstore 4
160 goto 171
---------------------------------------------------
163 aload 8
165 invokestatic #46 <Method WCETAn.
       Identify_Code(Lwcetan/Label;)V>
168 fload_3
169 fstore 4
---------------------------------------------------
171 aload 7
173 getstatic #17 <Field use_pow.Power_A Lwcetan/Mode;>
176 invokestatic #61 <Method WCETAn.
       Dead_Path(Lwcetan/Label;Lwcetan/Mode;)V>
179 aload 6
181 getstatic #17 <Field use_pow.Power_A Lwcetan/Mode;>
184 invokestatic #61 <Method WCETAn.
       Dead_Path(Lwcetan/Label;Lwcetan/Mode;)V>
187 aload 7
189 getstatic #19 <Field use_pow.Power_B Lwcetan/Mode;>
192 invokestatic #61 <Method WCETAn.
       Dead_Path(Lwcetan/Label;Lwcetan/Mode;)V>
195 aload 6
197 getstatic #19 <Field use_pow.Power_B Lwcetan/Mode;>
200 invokestatic #61 <Method WCETAn.
       Dead_Path(Lwcetan/Label;Lwcetan/Mode;)V>
203 fload 4
205 freturn }
```

The graph of basic blocks follows. Variable numbers are shown in parenthesis.