Addressing Dynamic Dispatching Issues in WCET Analysis for Object-Oriented Hard Real-Time Systems

Erik Yu-Shing Hu[*], Guillem Bernat and Andy Wellings

Real-Time Systems Research Group
Department of Computer Science
University of York, York, YO105DD, UK
{erik,bernat,andy}@cs.york.ac.uk

Abstract

There is a trend towards using object-oriented programming languages to develop hard real-time applications. However, some object-oriented features, such as dynamic dispatching and dynamic loading, are prohibited from being used in hard real-time systems because they are either unpredictable and/or unanalysable. Arguably, these restrictions could make applications very limited and unrealistic since they could eliminate the major advantages of object-oriented programming. This paper demonstrates how we can address the dynamic dispatching issues in Worst-Case Execution Timing (WCET) analysis with minimum annotations. The major contributions include: discussing the major issues involved in using and restricting dynamic binding features; weakening the restriction of using dynamic dispatching; presenting how to estimate tighter and safer WCET value in object-oriented hard real-time systems. Our approach shows that allowing the use of dynamic dispatching not only can provide a more flexible way to develop object-oriented hard real-time applications, but it also does not necessarily result in unpredictable timing analysis.

Keywords : Java, Hard Real-Time Systems, Real-Time Java, Worst-Case Execution Time (WCET) Analysis, Object-Oriented WCET, Dynamic Dispatching

1. Introduction

The success of hard real-time systems relies upon their capability of producing functionally correct results within defined timing constraints. In order to achieve this, it is of vital importance to guarantee that all hard real-time threads will finish their tasks within their deadlines. Typically, most scheduling algorithms assume that the WCET (Worst-Case Execution Time) of each task is known prior to doing the schedulability analysis.

The purpose of WCET analysis is to determine the maximum possible execution time that a piece of code may take. This analysis has to be safe; no underestimation of this value is allowed, but it should also be tight. In order to achieve a tight estimate, both the program flow, such as loop iterations and infeasible paths, and the execution characteristics of the object code on the target system, such as instruction caches and pipelining, must be taken into account. On the whole, the WCET analysis technique may be divided into two levels: high-level analysis and low-level analysis. The role of the high-level analysis is to analyse possible program flows from the source program, without regard to the time for each atomic unit of flow. The role of the low-level analysis, however, is to determine the timing aspects of the hardware features. A number of research approaches [7, 13, 17, 15] have demonstrated how to estimate WCET at both levels. Given the high-level analysis and low-level analysis, the final WCET estimation can be calculated.

In general, most WCET analysis approaches are only considered in relation to procedural programming languages. In fact, doing WCET analysis on object-oriented programs must take into account additional dynamic features provided in object-oriented languages. For the most part, object-oriented languages provide three major features: encapsulation, inheritance, and polymorphism. In fact, some of these features may result in object-oriented applications being either unanalysable and/or unpredictable. In order to use object-oriented languages in hard real-time systems, most research approaches have prohibited using dynamic features, such as dynamic loading and dynamic dispatching. Notable exceptions include [11, 14]. However, they do not give a satisfactory solution to the problem of dynamic dispatching. For example, Persson and Hedin [14] have proposed providing a maximum time-bound for dynamic dispatching methods, but they do not mention how we can estimate these maximum time-bounds.

Dynamic dispatching issues have been considered in compiler techniques for a number of years [1, 2, 8, 9, 18]. Unfortunately, these approaches cannot be directly applied to WCET analysis since they are solely optimising dynamic binding and do not guarantee that all dynamic binding will be resolved before run-time. However, in WCET analysis for hard real-time systems, the execution time of every single method has to be known prior to executing it. Therefore, most approaches in the WCET analysis field have simply assumed that dynamic dispatching features should be prohibited. It is possible that these restrictions could make applications very limited and un-

[*]This work has been funded by the EPSRC under award number GR/M94113

realistic because they might eliminate the major advantages of object-oriented programming.

For the above reasons, we argue that the use of dynamic dispatching should be allowed in object-oriented hard real-time systems. In this paper, we propose minimum annotations to address dynamic dispatching. Furthermore, we show that the correctness of these annotations can be easily validated with a combination of optimisation techniques, such as Class Hierarchy Analysis (CHA)[8]. The major contributions of this paper are:

- Discussing the major issues involved in restricting and using dynamic binding
- Weakening the restrictions by allowing dynamic dispatching
- Presenting how to estimate tighter and safer WCET values in object-oriented hard real-time systems
- Demonstrating how to fit this approach into a framework for real-time system development on Java (XRTJ)

The rest of the paper is organised as follows. Section 2 gives a brief review of WCET analysis, a summary of portable WCET analysis, and our framework for developing real-time programs in Java (XRTJ). Section 3 discusses the major issues connected with using and restricting dynamic binding features in object-oriented hard real-time systems. Following this, section 4 introduces annotations and shows how these annotations can be denoted in real-time applications in order to estimate tighter and safer WCET values. Section 5 gives an example to demonstrate how we can address dynamic dispatching issues and how safe and tight WCET estimations can be calculated. Finally, conclusions and future work are presented in section 6.

2. Background

2.1. WCET (Worst-Case Execution Timing) Analysis

There exists a well-founded ground of research on platform-dependent WCET analysis [7, 13, 17, 15]. 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[1] from the source code and the generated object code. The timing of each basic block is determined, for instance, by adding up the worst-case execution time of each of the machine instructions within the basic block. Loops are identified and annotated with the maximum number of iterations.

The timing information of each basic block can then be combined to determine the WCET of the whole program by using a simple timing schema [17], which is a set of rules to collapse the control flow graph annotated with timing information. Let WCET(S) denote the worst-case execution time of a code segment S, and assume that we initially have 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, essentially, the following rules [17]:

- $WCET(S_1; S_2) := WCET(S_1) + WCET(S_2)$
- $WCET(if\ E\ then\ S_1;\ else\ S_2) := WCET(E) + max(WCET(S_1), WCET(S_2))$
  where E is the condition expression 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 a bounded number of iterations.

The effects of low-level features such as 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 developers to define maximum loop bounds that could not be determined automatically, or provide more sophisticated information like mutually exclusive paths [7] or dependencies between loop induction variables [4]. These annotations are usually included in the source code as specially formatted comments and they are extracted by WCET analysis tools.

2.2. Portable WCET Analysis

A portable WCET analysis approach based on the Java architecture has been proposed by Bernat et. al. [5] within the Javalin project. It has been extended by Bate et al. [3] to address low-level analysis issues. The portable WCET analysis uses a three-step approach. The analysis assumes that the programmer has already compiled the Java source into a class file with a Java compiler.

The first step is the high level analysis. At this stage, the technique analyses annotated Java class files (JCF). Annotations are expressed by calls to a predefined static class WCETAn [5] and portable WCET information is computed in the form of so-called WCEF (Worst-Case Execution Frequency) vectors. WCEF vectors [3] represent execution-frequency information about 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 target Virtual Machines (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. This stage performs platform-dependent analysis (i.e. in the context of specific hardware and VM) of each Java byte code (JBC) instruction implementation. During this stage, information about the potential effects of pipelines and caches may be captured.

[1] A basic block 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.

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 analysed code sections. As the resources used by this stage are manageable, the calculations can easily be performed even on small virtual machines.

2.3. XRTJ Overview

Our previous work, called XRTJ (Extended Real-Time Java), extends the current Real-Time Java architecture [6] proposed by the Real-Time Java Expert Group. The XRTJ architecture has been developed with the whole software development process in mind: from the design phase to run-time phases. For example, using our approach, the system can be evaluated during the design, and the application's timing constraints can be validated during run-time. A summary of our previous work presented in [12] is given as below.

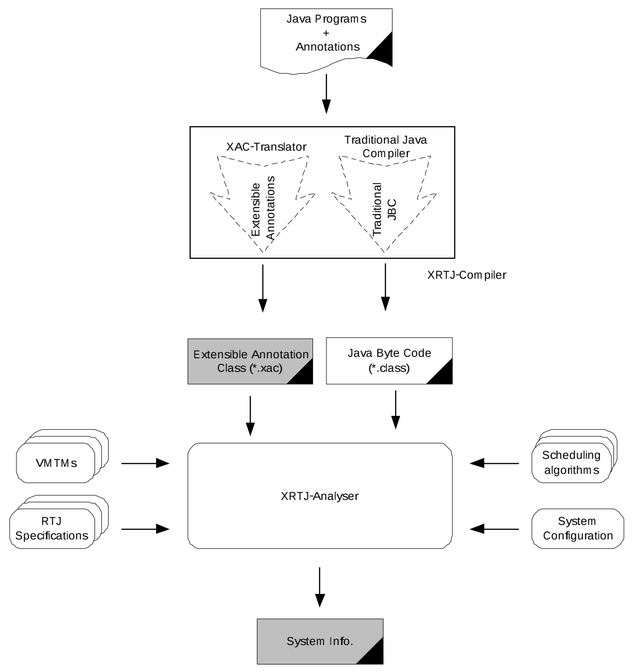

Figure 1. A block diagram of the XRTJ architecture for static timing analysis

As shown in Figure 1, the XRTJ architecture includes an Extensible Annotation Class (XAC) file, integration interfaces for real-time Java specifications, VMTMs (Virtual Machine Timing Models) and scheduling algorithms. Annotated Java programs are compiled into JCF (Java Class File) and XAC files by either a simple translator and a traditional Java compiler or an annotation-aware compiler. The VMTM is a timing model for the target virtual machine including a list of the WCET of native methods and JBC instructions, and the gain factors due to the favourable effects of pipelines across instruction sequences. The RTJ (Real-Time Java) specification interfaces are the abstract description of the specifications. These are used in the system to describe how the XRTJ-Analyser can extract the real-time parameters from the Java byte codes. The system configuration is the only input file provided by the developers, and defines the run-time environment of the system to be analysed in the XRTJ-Analyser. In accordance with the system configuration, the XRTJ-Analyser gets the information about the VMTM, the format of the RTJ specification, and scheduling algorithms and carries out the static timing analysis. An overview of the static timing analysis of the XRTJ approach is as follows.

Java programs are extended with timing annotations, such as maximum loop bounds. These annotations are extracted by the XAC translator or compiler, and the result is a compact representation of these annotations as an XAC file. The XAC files, together with the JCFs, are used by the XRTJ-Analyser to perform high-level WCET analysis, including gathering bounded loops and analysing flow information. We can use the VM timing model and flow information to carry out low-level WCET analysis. Given the high-level analysis and low-level analysis, WCET estimates can finally be calculated. Based on the real-time Java specification interface, the XRTJ-Analyser can gather real-time thread parameters, such as period and deadline, from the applications. Given estimated WCET values, real-time parameters, system configuration, and scheduling algorithms, the XRTJ-Analyser can carry out schedulability analysis for the whole system. Finally, the XRTJ-Analyser produces an output file for the timing analysis of the whole system. This information can then be used by the developers to refine the system.

2.4. XAC (Extensible Annotation Class File)

The XAC (Extensible Annotations Class) file stores extra information which cannot be expressed in the source code. XAC is an annotation structure that can be stored in files or as an additional code attribute in JCF. The XAC file has been designed with two main goals in mind: *portability* and *extensibility*. We have introduced a number of annotations in order to capture information for timing analysis in [12]. In this paper, we proposed additional annotations, which have the same format as our previous annotations and which can be translated into the XAC file, for addressing dynamic dispatching. This will be discussed in detail in section 4.

3. Dynamic Dispatching Issues

This section is mainly concerned with the major issues connected with using and restricting dynamic dispatching features. In this paper, these issues will be mainly discussed with the Java architecture, but our approach can easily apply to any object-oriented programming language.

Java distinguishes itself from other general-purpose object-oriented languages by its portability, networking, memory management, concurrent programming, and security features. However, the run-time characteristics (i.e. high frequency of method invocations and dynamic dispatches) make Java more difficult than other object-oriented programming languages, such as C++, for optimisation at compilation time. Therefore, addressing dynamic dispatching on the Java architecture is a big challenge and very interesting research topic in either dynamic compilation or WCET analysis. The following subsections discuss the major issues connected with restricting and using the dynamic dispatching feature in Java.

3.1. Issues connected with Restricting Dynamic Dispatching Features

From the Java virtual machine point of view, methods can be mainly divided into *Java methods* and *native methods*. Invoking a *Java method* in the Java virtual machine (JVM) creates a new stack frame, whereas invoking a *native method* does not push a new stack frame [19]. *Native methods* are not discussed here since they are outside the scope of this paper.

For the most part, *Java methods* may be classified into two main groups: *class (static) methods*, and *instance methods*. A *class (static) method* is a method which does not need an instance to be invoked, whereas an *instance method* requires an instance before it can be invoked. In other words, *class (static) methods* are invoked based on the type of the object reference, whereas *instance methods* are invoked based on the actual object [19]. In general, the *class methods* are translated into *invokestatic* Java byte code instructions, which uses static binding technique at run-time. By definition, a *static method* cannot be overridden and therefore no run-time dispatching is required. In contrast, *instance methods* are translated into the *invokevirtual* Java byte code instruction, which uses dynamic binding. Note that, although *instance methods* are normally invoked with *invokevirtual*, in specific situations two other Java byte code instructions may be used: *invokespecial* and *invokeinterface*. The *invokespecial* instruction is applied for instance initialisation, private methods, and methods invoked with the *super* keyword. It differs from *invokevirtual* in the manner in which it uses static binding. The *invokeinterface* performs the function as *invokevirtual*, but it is used solely when the type of reference is an *interface*. In the following sections, we use *'instance method'* to mean those methods that are translated into either *invokevirtual* or *invokeinterface* instructions and which may be overridden by child classes.

In the Java language semantic, *Java methods* may be defined as: *public*, *private*, *protected*, *static*, and *final*. By definition, *private*, *static*, and *final* methods cannot be overridden by any other classes, and only *public* and *protected* methods can be overridden by child classes. On the whole, the dynamic features of objected-oriented, such as inheritance and overriding, are offered in the *instance methods*, which are defined as *public* and *protected* in Java. Therefore, if one prohibited the use of dynamic binding features in object-oriented real-time applications, only *static*, *private*, and *final* methods could be used. Obviously, these restrictions could eliminate the major advantages of object-oriented programming. Arguably, these kinds of applications no longer appear object-oriented and have even less expressive power than procedural languages in terms of reusability and extensibility. An alternative approach is to force the programmer only to use static binding. In Java, this can be achieved by disallowing assignment and parameter association between objects in the same class hierarchy. For the above reasons, we conclude that dynamic dispatching should be allowed in hard real-time systems in an appropriate way.

3.2. Issues involved in Using Dynamic Dispatching Features

Unlike procedural programming languages, the analysis of object-oriented programming in WCET analysis needs to consider more dynamic characteristics. In real-time systems, all hard real-time threads not only have to be analysable and predictable, but also have to meet their deadline at run-time. In fact, using the dynamic binding features in Java applications may lead *instance methods* to be either unanalysable and/or unpredictable.

```
class  A {
   //  WCET:100ms
  public  void  m1() {
   ...
  }
   ...
}

class  B extends A {
   //  WCET:25ms
  public  void  m1() {
   ....
  }
   ...
}

class  C extends A {
   //  WCET:200ms
  public  void  m1() {
   ....
  }
   ...
}
```

```
class  App {
  Call_m1(A ax) {
     //  dynamic dispatching  occur
    ax.m1();
  }
  public  static  void  main ( String []  args ) {
    A a= new A();
    B b= new B();
    C c= new C();
     ...
    Call_ml(a);
    Call_m1(b);
    Call_m1(c);
     ...
    if (x>5)
      a = c;
    else
      a = b;
     ...
    Call_m1(a);
  }
}
```

Figure 2. Example 1

For example, in Figure 2, class A is a parent class and has a public method called m1(). Then, class B and class C extend the class A and override the method m1(). Considering the Java byte code in Figure 2, we can see that the exact method of the ax.m1() in the Call_m1() method is unknown until run-time since it uses dynamic binding features. The WCET values for A.m1(), B.m1() and C.m1() are 100ms, 25ms, and 200ms respectively. In this situation, if we estimate the WCET value of ax.m1() with A.m1() method, it is very pessimistic if the instance type is B, or it is even unsafe if the instance type is C. Again, from the source codes, it can be observed that the WCET of the first call Call_m1(a) and the last call Call_m1(a) are different since an instance of parent class can denote an instance of any descendant of the class.

Apparently, using dynamic dispatching in a hard real-time system may result in the whole system being not only unpredictable and unanalysable, but also either unacceptably pessimistic or unsafe. Therefore, every single *instance method* should be analysed carefully if dynamic dispatching features are allowed.

$//@\ \mathbf{WCET\_Label}(Full\_Class\_Name.Method\_Name(argument\ types),\ Pattern\_Format)$	....	A1
$//@\ \mathbf{UseWCET}(Full\_Class\_Name.Method\_Name(argument\ types),\ Pattern\_Format)$	....	A2
$//@\ \mathbf{DefineScope}(Scope\_Name)$	....	A3
$//@\ \mathbf{ScopeWCET}(Scope\_Name,\ nCount * UseWCET(...)\ + ...)$	....	A4
$//@\ \mathbf{maxWCET}(\&Full\_Class\_Name.Method(argument\ types)\ \ -/+$ $Full\_Class\_Name/\ \&Full\_Class\_Name,\ Pattern\_Format)$	....	A5

Table 1. WCET annotations

4. Dynamic Dispatching with Annotations

In our previous work [12], we use the $//@Mode(...)$ annotation, proposed by Chapman et al. [7], to help estimate tighter WCET values for different execution times on a specific iteration loop or function call. Similarly, in object-oriented programming languages, the execution time of either *class* or *instance method* may vary. In addition to this, in Java, the execution time of a particular *instance method* may be different in its descendant classes if child classes override the method. Therefore, it is clear that the WCET analysis will be complicated if dynamic binding features are taken into account.

In our approach, we assume that the source code of all hard real-time thread is analysable. This section introduces minimum annotations, using the XAC (Extensible Annotation Class) format presented in [12]. These annotations can not only address the dynamic dispatching feature, but can also offer WCET analysis that achieves tighter and safer WCET estimation. All annotations are given in Table 1 and each one is discussed in the following subsections.

4.1. WCET Annotations for Object-Oriented Methods

As shown in Figure 2, the type of the instance object 'a', which originally has the type of Class A, was changed to the type of B object or C object after the *if* statement. Most of these dynamic type changes can be analysed by current compiler optimisation approaches, such as Class Hierarchy Analysis (CHA) [8] and Rapid Type Analysis (RTA) [2], for dynamic compilation. However, the dynamic compilation approaches are solely for optimising dynamic binding and they do not guarantee that all dynamic binding will be resolved before run-time. In fact, in WCET analysis for hard real-time systems, the execution time of every single *instance method* has to be known prior to executing it.

Moreover, in procedural programming languages, the relationships between functions or procedures are relatively simple. They have one call hierarchy and neither inheritance nor polymorphism is supported. Unfortunately, in object-oriented languages, the naming of methods is relatively difficult to understand and analyse if several objects are using the same method name, for example using overridding and overloading. As a result, well-structured annotations need to be considered in relation to the class hierarchy information in object-oriented languages.

Our approach is to provide minimum annotations to ensure the predictability of dynamic binding methods statically and estimate safe and tight WCET for hard real-time applications. In this section, two major WCET annotations (A1 and A2), which offer expressive power to cope with object-oriented features, are introduced in order to address dynamic dispatching problems. The *WCET_Label()* annotation (A1) is provided for making a label for a specific mode for particular executing characteristics in either class or instance methods, whereas *UseWCET()* annotation (A2) is offered to denote a specific mode or method in the applications. The complete format of these two annotations is given in Table 1. In both annotations, *Pattern_Format* may denote a *path, label, mode, symbolic mode with input data ranges or none*. Using the *Pattern_Format*, we can define a specific path, label, mode, or symbolic mode for different WCET estimations for the method. In the symbolic mode, we can express the input data range (DR) as follows:

Data Range	Notation
$a \le DR \le b,\ where\ a < b$	[a..b]
$a < DR < b,\ where\ a < b$	(a..b)
$a \le DR < b,\ where\ a < b$	[a..b)
$a < DR \le b,\ where\ a < b$	(a..b]
$-\infty < DR < b$	(-inf..b)
$a < DR < \infty$	(a..+inf)

Table 2. Symbolic input data range format

In addition, in order to take into account the polymorphism features, argument types need to be considered. In a combination of the full class name, method name and argument types, the full name for each method can be defined in the annotations. A simple example is given in Figure 3.

4.2. WCET Annotations for Nested Scopes in Objected-Oriented Programs

Considering the situation of the program in Figure 3, it is clear that the annotations proposed in the previous section are not tight enough to represent complicated structures or nested loops. The example shows that it could be pessimistic or unsafe if this situation occurred. To represent the dynamic behaviour of a program additional annotations, which can represent the WCET estimation for a specific scope, are necessary. Furthermore, in order to provide a more flexible environment for the object-oriented hard real-time applications, we need to consider more complicated conditions, such as nested loops.

In this section, two additional WCET annotations (A3 and

```
class A {
  //@ WCET_Label(A.m1())
  public void m1 () { ... }   // given 100ms
}

class B extends A {
  //@ WCET_Label(B.m1())
  public void m1 () { .... }   // given 25ms
}

class App {
  public static void main ( String [] args ) {
     A a= new A();
     B b= new B();
      ...
     //@ DefineScope(ForScope)
     for(int i=0;i<5;i++) {
        if(i==2) // type changing
           a = b;
        //@ ScopeWCET(ForScope,2*UseWCET(A.m1)+3*UseWCET(B.m1))
        a.m1();
     }
  }
}
```

Figure 3. Example 2

A4), which address nested scopes or loops, are introduced. The *DefineScope()* annotation (A3) is provided for defining a simple or nested loop to establish the WCET, whereas the *ScopeWCET()* annotation (A4) is offered to denote the WCET estimation for the whole specific scope. These two annotations are given in Table 1.

The scope annotations, which have been used in procedural programming languages [10], are introduced in order to represent a particular scope or nested loops in WCET analysis. Using these scope annotations, we can achieve very tight and safe WCET for a specific scope or nested loops.

4.3. Maximum WCET Annotations for Dynamic Dispatching

Using annotations presented in previous sections (Section 4.1 and 4.2), we can address most issues involved when using dynamic dispatching features. However, the drawback is that these annotations need to know exactly which methods are going to be invoked at run-time. Unfortunately, it could be possible that there is more than one particular method that may be invoked.

In this section, the *maxWCET(...)* annotation (A5), which can denote a set of complicated class hierarchy, is introduced. This annotation can suggest that the WCET of a dispatching method should be considered to be the maximum WCET of the class family[2] containing that method. Subsets of the class family can also be specified. One of the major differences with other annotations is that the *maxWCET(...)* supports reusability and extendibility features. This can be achieved by the support of an annotation-aware compiler.

In the *maxWCET(...)* annotation (A5), ”&”of A denotes the whole class family of A. In this annotations, we can use ”+” and ”-” to express the union or subtraction of a single class or a class family for the method. The definition of the *Pattern_Format* is described in section 4.1. Given class A has two descendant classes B and C, and B and C each has a number of descendant classes, if we would like to denote a method (A.mx()) of the class A, in which we do not want to consider the class C family, we can use the *maxWCET(...)* as below.
e.g. $//@\ maxWCET(\&A.mx() - \&C)$

[2]A class family of a class is a set of the classes including the class itself and all the child classes inherited from it.

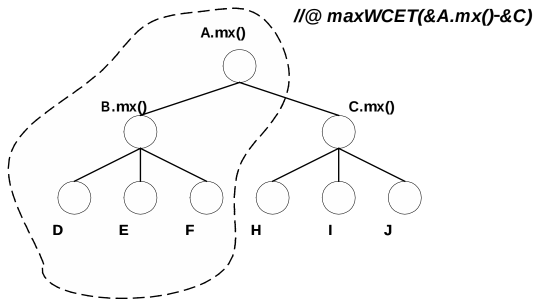

Figure 4. maxWCET(...) annotation

The concept of these annotations is similar to CHA [8], which is used for optimisation in compilation techniques, whereas here the annotation may provide tighter specification for a specific method. If we apply the CHA approach, the WCET value could be very pessimistic. Again, using Figure 2, here we only need to consider the execution of A.m1() from the WCET value either A.m1() or B.m1(). If we used the CHA approach or other WCET approaches for the WCET value for the A.m1() with max(A.m1(), B.m1(), C.m1()), the estimation is very pessimistic. It is clear that using $//@\ maxWCET(...)$ annotation can not only address these issues and but can also achieve tighter and safer WCET estimation.

5. Evaluations

This section gives an example to demonstrate how we can address dynamic dispatching issues and how safe and tight WCET estimations can be calculated. In addition, this section discusses how we can validate the correctness of these annotations.

5.1. An Example

We use an example which is part of the sensor control system of an aircraft control system to discuss how we can provide safe and tight WCET annotations in the parent classes and how to use them in the child classes. The semantics and real-time APIs (Application Program Interfaces) used in the example are in line with the RTSJ specification [6]. In Figure 5, it can be observed that an abstract Sensor class has three subclasses: *Temperature_Sen* class, *Pressure_Sen* class and *Speed_Sen* class. The *Temperature_Sen* class has three child classes: *AirTempSen* class, *JetEngineTempSen* class and *LandingDeviceTempSen* class.

The purpose of these classes is to detect the surrounding environment, such as air temperature, Jet-engine temperature, and

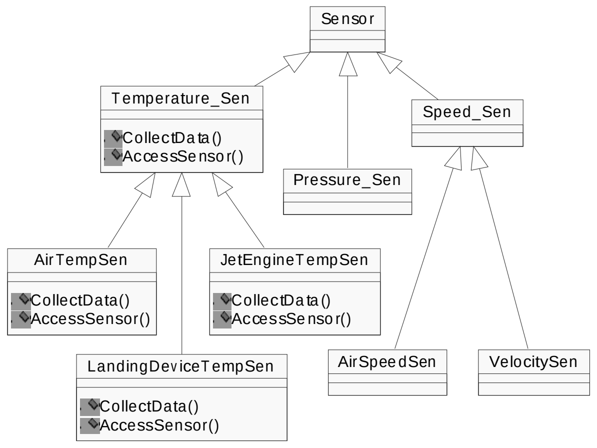

Figure 5. A class hierarchy for Sensor Control Systems

landing devices' temperature, of the aircraft and then to report temperature information to related device objects or systems. Similarly, the *AirSpeedSen* class and *VelocitySen* class, which are inherited from the *Speed_Sen* class, detect the air speed and velocity of the aircraft respectively. The limited space of this paper means that we can only discuss the *Temperature_Sen* class in this section. Moreover, in order more easily to understand our approach, in the example, we only consider the high-level timing analysis of the program and produce the WCET estimation. Of course, in reality, we have to consider the low-level timing analysis to calculate the tight and safe WCET estimations.

```
class Temperature_Sen extends Sensor {
  ...
  //@ WCET_Label(Temperature_Sen.CollectData(V))  // WCET:100ms
  //@ WCET_Label(Temperature_Sen.CollectData(V), TakeOff_Mode) //WCET:200ms
  public int CollectData () {  // Overridden by children  classes
    ...
    return  result ;
  }
  //@ WCET_Label(Temperature_Sen.AccessSensor(V)) //WCET:200ms
  public int AccessSensor () {  // Overridden by children  classes
    ...
    return  result ;
  }
  ...
}
```

Figure 6. A fragment of the Temperature_Sen Java program

As shown in Figure 6, the *Temperature_Sen* class has *CollectData(...)* and *AccessSensor(...)* methods. The WCET annotations can be added to analyse and validate the design and WCET behaviour of the *Temperature_Sen* class as follows. In this object, we add two annotations for *CollectData(...)* method: one is the default *//@WCET_Label(...)* and the other is *TakeOff_Mode*. We assume that the WCET values for the default *//@WCET_Label(...)* and the *TakeOff_Mode* are 100ms and 200ms respectively. In a similar way, we can analyse other methods either in parent or in child classes. In the child classes, we assume that the WCET values, denoted with default *//@WCET_Label(...)*, for *CollectData(...)* method in the *AirTempSen*, *JetEngineTempSen*, and *LandingDeviceTempSen* are 110ms, 120ms and 130 respectively. The WCET values for the *TakeOff_Mode* are twice the WCET value of the default *//@WCET_Label(...)* in the same class in each child class. Moreover, the WCET values for *AccessSensor(...)* method in the *AirTempSen*, *JetEngineTempSen*, and *LandingDeviceTempSen* are 210ms, 220ms and 230 respectively. Here we shall have to omit the source codes of these child classes.

```
import javax. realtime .*

 static  AirTempSen ATTempSen;
 static  JetEngineTempSen JETempSen;

class public SensorController extends RealTimeThread  ( .... ) {
  Temperature_Sen TempSen;
   ...
   SensorController ( PriorityParameters  schP , PeriodicParameters  relP ) {
     super(schP , relP );
     TempSen = JETempSen; //  default  with  JetEngineTempSen
      ...
  }
  private void Call_Sensor (Temperature_Sen callTempSen) {
      ...
      // With design knowledge, we can denote the maxWCET(...) as below.
     //@ maxWCET(&Temperature_Sen.AccessSensor− &LandingDevice_TempSen)
      //  the  WCET is max (200, 210, 220) => WCET: 220ms
     callTempSen.AccessSensor ();
      ...
      //  given  the  rest  of  the  WCET is 100ms
  }
  public void run () {
     while( waitforNextPeriod ());
         ...
       //@ UseWCET(JetEngineTempSen.AccessSensor(V)) //WCET:220ms
       TempSen.AccessSensor()
        ...
        Call_Sensor (TempSen); // 220 ms+100ms
       TempSen = ATTempSen; // given 7 ms  //  type  changing occur
                ...
       //@ UseWCET(AirTempSen.AccessSensor(V)) //WCET:210ms
       TempSen.AccessSenor();
        ...

       //@ UseWCET(AirTempSen.CollectData(V),TakeOff_Mode) //WCET:220ms
       TempSen.CollectData ();
        ...
       TempSen = JETempSen; // given  7  ms  //  type  changing occur
        //  given  the  rest  of  the  WCET is 200ms
     }
  }
}
```

Figure 7. A fragment of the SensorController Java program

Then, as shown in the Figure 7, we can analyse the real-time thread (*SensorController*), which is defined as a *PeriodicThread* in accordance with the RTSJ. In the *SensorController* object, assume that we only need to consider the *AirTempSen* object and *JetEngineTempSen* object. Therefore, with design knowledge, we can add an //@maxWCET(&Temperature_Sen.CollectData() - &LandingDevice) annotation for *TempSen.CollectData()* code to achieve tighter WCET estimation in the Call_Sensor() method. Therefore, for this code, we can assume that the WCET value is 220ms. In the run() method, analysing the source code, we can denote two *//@UseWCET(...)* annotations. Finally, we can calculate a tight and safe WCET value for the *SensorController* periodic real-time thread. The WCET value for the run method in the *SensorController* thread can be calculated as follows:

WCET(SensorController.run ())
= 220ms+ (220ms+100ms)+ 7ms+ 210ms+ 220ms+ 7ms+ 200ms
= 1184ms

5.2. Correctness of Annotation

It is an open question for most annotation-based approaches as to how to validate if the provided annotations are correct or not. Obviously, combining the optimisation techniques, such as CHA [8] or RTA [2], with our approach the annotations can be simply verified, if there is no dynamic linking at run-time. For example, applying the CHA approach, we can easily get the maximum bound of the classes hierarchy information from the Java byte code. In addition, since the *//@ maxWCET(...)* annotations defined in the parent classes can be inherited from child classes, the annotation defined in the parent classes can be used to validate the annotations added in the child classes. Therefore, using these annotations, the object-oriented hard real-time systems not only can achieve tight and safe WCET estimations, but can also validate the correctness of the system's design and WCET annotations.

6. Conclusions and Future Work

This paper has explored the ways in which dynamic dispatching can be addressed in object-oriented hard real-time systems. Our approach shows that allowing the use of dynamic dispatching not only can provide a more flexible way to develop object-oriented hard real-time applications, but it also does not necessarily result in unpredictable timing analysis. Moreover, it demonstrates how to achieve tighter and safer WCET estimations in object-oriented real-time applications.

Currently we are working on the XRTJ architecture [12] to improve the timing analysis to take into account other dynamic features of the object-oriented languages, such as memory management, dynamic loading, and remote method invocation (RMI). We are also assessing our timing analysis approach with the aim of providing information about the dynamic behaviour of object-oriented hard real-time threads to improve the performance of aperiodic threads at run-time. Furthermore, validating annotations and integrating this approach into dynamic timing analysis will be a major feature of our future work.

References

[1] G. Aigner and U. Holzle. Eliminating Virtual Function Calls in C++ Programs. *ECOOP' 96 Conference Proceedings*, Springer Verlag LNCS 1098:142–166, 1996.

[2] D. Bacon and P. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. *Proceedings of the ACM Conference on Obejct-oriented Programming Systems, Languages, and Applications (OOPSLA'96)*, October 1996. San Jose, California.

[3] I. Bate, G. Bernat, G. Murphy, and P. Puschner. Low-Level Analysis of a Portable Java Byte Code WCET Analysis Framework. *In 6th IEEE Real-Time Computing Systems and Applications (RTCSA2000)*, pages 39–48, December 2000. South Korea.

[4] G. Bernat and A. Burns. An Approach to Symbolic Worst-Case Execution Time Analysis. *In 25th IFAC Workshop on Real-Time Programming*, May 2000. Palma (Spain).

[5] 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*, pages 81–88, June 2000.

[6] G. Bollella, J. Gosling, and B. Brogdol. *Real-Time Specification for Java*. Addison Wesley, 2000. http://www.rtj.org.

[7] 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.

[8] J. Dean, D. Grove, and C. Chambers. Optimisation of Object-Oriented programs using Static Class Hierarchy Analysis. *ECOOP' 95 Conference Proceedings*, Springer Verlag LNCS 952:77–101, 1995.

[9] D. Detlefs and O. Agesen. Inlining of Virtual Methods. *ECOOP' 99 Conference Proceedings*, Springer Verlag LNCS 1628:258–277, 1999.

[10] J. Engblom and A. Ermedahl. Modeling Complex Flows for Worst-Case Execution Time Analysis. *In proceedings of the 21st IEEE Real-Time System Symposium (RTSS2000)*, December 2000. Orlando, Florida USA.

[11] J. Gustafsson. *Analysing Execution Time of Object-Oriented Programs with Abstract Interpretations*. Phd thesis, Department of Computer Systems, Information Technology, Uppsala University, Sweden, May 2000.

[12] E. Y.-S. Hu, G. Bernat, and A. Wellings. A Static Timing Analysis Environment Using Java Architecture for Safety Critical Real-Time Systems. *7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS-2002)*, pages 64–71, January 2002.

[13] 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.

[14] P. Persson and G. Hedin. An Interactive Environment for Real-Time Software Development. *Proceedings of the 33rd International Conference on Technology of Object-Oriented Languages (TOOLS Europe 2000)*, June 2000. St. Malo, France.

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

[16] P. Puschner and A. Wellings. A Profile for High-Integrity Real-Time Java Programs. *Proceedings of the 4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing ISORC 2001*, pages 15–22, 2001.

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

[18] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallee-Rai, P. Lam, E. Gagnon, and L. Hendren. Practical Virtual Method Call Resolution for Java. Sable Technical Report No. 1998-7, McGill University, Canada, 1998.

[19] B. Venner. *Inside the Java 2 Virtual Machine*. McGraw-Hill, 2nd. edition, 1999.