A Static Timing Analysis Environment Using Java Architecture for Safety Critical 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

Certainly, in hard real-time systems, it is reasonable to argue that no hard real-time threads should behave in an unpredictable way and that schedulability should be guaranteed before execution. In order to guarantee the timing constraints of portable code for hard real-time applications, a particular static timing analysis is necessary. In this paper, we provide a static timing analysis environment for the development of real-time applications on the Java architecture. The major contributions include introducing a novel Extensible Annotations Class (XAC) format to capture portable annotations from the source level, presenting how to integrate XACs with portable Worst-Case Execution Time (WCET) analysis, describing how to obtain real-time thread parameters from Real-Time Java's specifications, and demonstrating how static timing analysis using the Java architecture can be carried out from portable code.

Keywords

Java, Hard Real-Time Systems, Real-Time Java, Worst-Case Execution Time Analysis, Portable WCET Analysis

1. Introduction

The success of hard real-time systems relies upon their capability of producing functionally correct results within defined timing constraints. Clearly, it has to be guaranteed that all hard real-time tasks will meet their deadlines in line with the design. To guarantee this, predictability of the system and static timing analysis are of vital importance. Typically, most scheduling algorithms assume that the WCET (Worst-Case Execution Time) of each task is known prior to doing the schedulability analysis. Furthermore, the predictability of the system and WCET values enable developers to allocate more precise resources during the design phase.

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 also should 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, whereas the role of the low-level analysis is to determine the timing aspects of the hardware features. A number of research approaches [6,7,9,12,14,17] 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 approaches are tied to either a particular language or target architecture. Only the Javelin project [1,3], which presents how worst case execution time analysis can be performed on Java byte code and how portable timing annotations can be provide with Java byte code, is concerned with portability for WCET analysis. 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

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

flow and data flow information. Platform independent analysis is achieved by parameterising the different targets. There is some additional pessimism in performing the WCET process in this particular way, which compensates for the added benefits that portability brings [1,3].

Indeed, portability is a common requirement of future real-time applications. One of the most popular programming languages which supports high portability is Java. Even though the language was designed as a general-purpose object-oriented language, it distinguishes itself from other general-purpose object-oriented languages by its portability, networking, memory management, concurrent programming, and security features. The Java architecture includes Java language, Java Application Programming Interface (API), Java Virtual Machine (JVM), and Java Byte Code (JBC). Java supports portability with its Java byte codes and Java virtual machine. In fact, JBC can be produced not only from Java programs, but also from other programming languages supported by a specific compiler. Therefore, taking advantage of Java byte code, other programming languages can be easily migrated to the Java architecture. For instance, Ada programs can be translated into Java byte codes by either JGNAT [5] or the Aonix [13] compilers, and then executed upon a Java virtual machine.

Yet, the non-deterministic behaviour of memory management, poor performance of most Java implementations, and the lack of real-time facilities have hindered the acceptance of Java in real-time and embedded applications. In order to address these issues, two recent approaches have been attempted to provide real-time extensions to Java: Real-Time Specification for Java (RTSJ) [4] and the Real-Time Core extensions to Java [10]. These specifications have addressed the issues related to using Java in a real-time context, including scheduling support, memory management issues, interaction between non-real-time Java and real-time Java programs, and device management among others. However, none of the specifications provide a satisfactory solution for portable WCET analysis.

Since Java is an object-oriented programming language, in addition to portability it also supports other reusability. For the most part, object-oriented programming languages provide three major features: encapsulation, inheritance, and polymorphism. These features may lead object-oriented applications to be either unanalysable or unpredictable, or both. In order to use object-oriented programming languages in safety critical real-time systems, some language features need to be restricted.

A profile for high-integrity real-time Java programs based on the RTSJ specifications has been proposed by Puschner and Wellings [16]. The profile gives an overview of how to develop efficient applications whose temporal behaviour needs to be exactly predictable. They discuss necessary restrictions of the RTSJ and propose a realisation of the profile that meets the temporal requirements of high-integrity real-time systems. The paper [16] presents the restrictions inherent in the profile including threading model, inter-process communication and synchronisation, memory management, and the representation of time and clocks.

Certainly, in hard real-time systems, it is reasonable to argue that no hard real-time threads should behave in an unpredictable way and that schedulability should be guaranteed before execution. In order to guarantee the timing constraints of portable code for hard real-time applications, a particular static timing analysis is necessary.

To address these issues, we introduce a static timing analysis for the Java architecture in safety critical real-time applications. In our approach, we assume that the applications are at least analysable. In order to achieve this, the applications have to be used with the profile which is presented in [16]. The major aim of our approach is to provide a timing analysis environment for the development of hard real-time applications using the Java architecture. We introduce a novel extensible annotation class format, the so-called Extensible Annotations Class (XAC), to provide timing information from the design phase to static analysis or run-time phases. Using the XAC, we extend the work performed under the Javelin project [1,3] in terms of portable WCET analysis. This paper presents how static timing analysis using the Java architecture can be carried out.

The major contributions of this paper are:

- introducing a novel Extensible Annotations Class (XAC) structure
- presenting how to integrate the XAC with portable WCET analysis in detail
- describing how to gather real-time thread parameters from specifications
- demonstrating how static timing analysis using Java architecture can be carried out from portable code

The rest of the paper is organised as follows. Section 2 gives a brief review of WCET analysis and portable WCET analysis. Section 3 provides an overview of our architecture. The subsequent sections describe the components of the architecture in detail: section 4 presents a summary of our novel XAC structure, section 5 describes how to integrate them with the portable WCET analysis [1,3], section 6 describes our approach to integrating the Real-Time Java specifications [4,10], and section 7 presents how static timing analysis can be conducted on the Java architecture in our XRTJ-Analyser (Extended Real-

Time Java). Section 8 presents the current status and future work of our project. Finally, conclusions are presented in section 9.

2. Worst-Case Execution Time Analysis

This section gives a summary of the WCET (Worst-Case Execution Time) analysis and portable WCET analysis. There exists a well-founded ground of research on platform-dependent WCET analysis [6,7,9,12,14,17]. 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[†] 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 a 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, basically, the following rules:

- 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 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 a 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 developers to define maximum loop bounds that could not be determined automatically or more sophisticated information, like mutually exclusive paths [6] or dependencies between loop induction variables [2]. These annotations are usually included in the source code as specially formatted comments and they are extracted by WCET analysis tools.

2.1 Portable WCET Analysis

The portable WCET analysis using Java byte code (the Javelin project) has been proposed by Bernat et. al. [3]. It has been extended by Bate et al. [1] 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. In this stage, the technique analyses annotated JBC. Annotations are expressed by calls to a predefined static class WCETAn [3] and portable WCET information is computed in the form of so-called WCEF (Worst-Case Execution Frequency) vectors. WCEF vectors [1] 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 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 JBC instruction implementation. In this stage, information about the 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 analysed code sections. As the resources used by this stage are manageable, the calculations can easily be performed even on small virtual machines.

3. Framework Overview

Our approach, called XRTJ (Extended Real-Time Java), extends the current Real-Time Java architecture [4] that is 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

[†] 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.

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. This paper is mainly concerned with the static timing analysis part. Dynamic timing analysis will be discussed in future work. In this paper, we use Java programs to discuss our architecture. Indeed, other programming languages using a particular compiler, which may translate into Java byte code, can also migrate to our architecture easily.

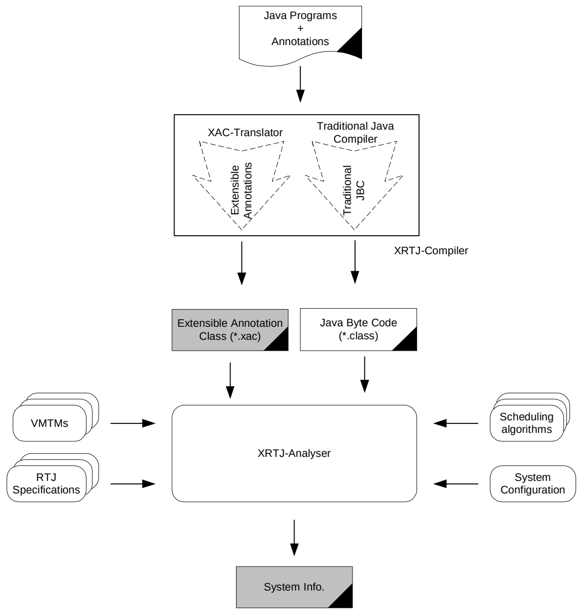


Figure 1. A block diagram of the XRTJ architecture for static timing analysis

A block diagram, which illustrates the XRTJ architecture in terms of static timing analysis, is given in figure 1. In the figure, we can see that XRTJ architecture includes the Extensible Annotation Class (XAC) file, integration interfaces for real-time Java specifications, VMTMs (Virtual Machine Timing Model) and scheduling algorithms. As shown in figure 1, 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 which 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 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 given below.

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.

4. XAC (Extensible Annotation Class File)

The XAC (Extensible Annotations Class) file stores extra information which cannot be expressed in the source code. This section is mainly concerned with how to store information about timing annotations in the XAC file. Although we sometimes use the term XAC file, it is not necessarily a physical file. XAC is an annotation structure that can be stored in files or as an additional code attribute in JCF.

4.1 Features

The XAC file has been designed with two main goals in mind: *portability* and *extensibility*.

Extensibility: The motivation for the XAC files is to be able to store extra information about the program source without changing the JBC format. However, it is relatively difficult to define a complete structure to provide enough information from the source code level for all purposes at once. We realise, therefore, that the specification for the XAC should be designed with extensibility in mind. For example, although we mainly focus on capturing information for timing analysis, the file could also be used to hold extra information needed for model checkers or any other tools.

Portability: As mentioned before, JBC may be generated from other programming languages supported by a specific compiler. Furthermore, JBCs are highly portable and can be executed on various platforms supported by a particular Java virtual machine. As a result of this, to support the Java architecture, XAC has to be designed with portability features. Moreover, we realise that the XAC file has to be designed with not only platform independence, but also language independence, in mind. Taking advantage of the portability of the XAC file, we can use the XAC file in either static or dynamic analysis tools.

4.2 Annotations Format

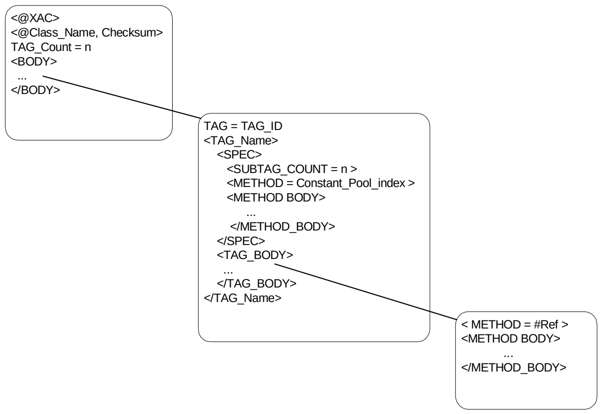
All annotations in our approach are introduced with the characters ' //@' for single line and '/*@ .... @*/' for multiple lines. These formats are assumed to be comments in Java. Similar structures of annotations are applied in the JML (Java Modelling Language) approach [11], and the ESC/Java (Compaq Extended Static Checker for Java) [8]. However, these projects mainly focus on recording detailed design decisions for a software module [11] or checking runtime errors by a modular program checker [8].

Even though the XAC file is provided for each class file in Java programs, its format is different from the JBC format. As all annotations are provided as comments in Java programs, these codes, including annotations, can be compiled by a traditional compiler to generate traditional JBC files and run on a traditional Java virtual machine. Figure 3 gives a fragment of code including the annotations.

4.3 XAC Format

Each XAC file is generated for a specific JCF (Java Class File). Therefore, the relationship between JBC and XAC is *one to one*. Since the XAC files are not defined for any particular operating systems, these files are easy to apply in annotation-aware tools or JVMs. Since the XAC file is designed for extensibility, the annotation formats of the XAC file may be variable. Therefore, all data structures need to be declared in the specification area.

The format of the XAC file is given in figure 2. In order to speed up the reading of these files, the XAC file can be expressed in binary format. The XAC file includes *class namespace*, *checksum*, *total number of tags*, *data format specification* and *contents of the annotations*. The *checksum* can help the JVMs to verify the consistency between the JBC file and the XAC file. It can also be used in both static and dynamic analyses. The outermost layer of the XAC file defines how many TAG are contained in this XAC file. Each TAG encompasses a specification and its body. The specification may declare the format of the contexts. The body includes the annotations whose formats are defined in the specification.



```
<@XAC>
<@Class_Name, Checksum>
TAG_Count = n
<BODY>
  ...
</BODY>
```

```
TAG = TAG_ID
<TAG_Name>
   <SPEC>
     <SUBTAG_COUNT = n >
     <METHOD = Constant_Pool_index >
     <METHOD BODY>
          ...
      </METHOD_BODY>
   </SPEC>
   <TAG_BODY>
     ...
   </TAG_BODY>
</TAG_Name>
```

```
< METHOD = #Ref >
<METHOD BODY>
         ...
</METHOD_BODY>
```

Figure 2: The format of the XAC file

5. Integrating Portable WCET Analysis

This section shows how the XRTJ approach can be integrated with portable WCET analysis. To support WCET, analysis annotations have to be provided in the source code. Bernat et al. [3] present a high-level WCET analysis technique using JBC to provide WCET annotations by means of a WCETAn class. It has sufficient capabilities to provide WCET annotations from the source level to JBC. The approach has showed how to extract data flow and control flow information from the JBC program without relying on the source code. The WCETAn class allows scope or code-block boundaries, the maximum number of iterations of loops, execution modes and arbitrary path execution frequencies to be described [15]. However, the WCETAn approach requires either a tool or WCETAn-aware JVM to eliminate these annotation instructions from JBC before executing.

We extend the WCETAn approach to accumulate timing information from the source level. We introduce the XAC (Extensible Annotation Class) file to provide high-level WCET annotations instead of using WCETAn Class. We support similar annotations to the one used by the Javelin project [3]. These are given in table 1.

Take the following code example in figure 3, which shows how to specify that a loop iterates 50 times in the worst case, but that in the particular modes of operation called *Quick_Mode* and *Normal_Mode* they only iterate 10 and 30 times respectively.

Table 1. WCET Annotations

Type	***Annotation***	***Description***
WCET Tags	//@ Mode (Mode_name)	Build a mode
	//@ Label (Label_name)	Build a label
Naming Tags	//@ Define_Mode (Mode_name)	Define a mode
	//@ Use_Mode (Mode_name)	Use the specific mode
	//@ Identify_Code (Label_name)	Identify a label
Assertions	//@ Loopcount (Max_loop_count)	Maximum loop count
	//@ Loopcount (Max_loop_count, Mode_name)	Maximum loop count associated with the mode
	//@ Dead_Path (Mode_name, Label_name)	Infeasible path
	//@ Begin_WCET (Label_name)	The beginning of the WCET associated with the label
	//@ End_WCET (Label_name)	The end of the WCET associated with the label

```
...
public void Call_ForLoop(int nDoLoop) {
          ...
          //@ Mode( Quick_Mode )
          //@ Mode( Normal_Mode );
          ...
          //@ Loopcount( 50 );
          //@ Loopcount( 10, Quick_Mode );
          //@ Loopcount( 30, Normal_Mode );
          for(i=0;i<nDoLoop;i++) {
             ...
          }
          ...
}
...
```

Figure 3. A fragment of the annotated *Call_ForLoop* method

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 that all calls use the maximum number of iterations. Using an XAC translator or compiler, the XAC file can be produced from the annotated Java program. The text format of the XAC file is given in figure 4. In the figure, we can see that four types of WCET annotation formats are defined in the specifications. Each annotation has a unique identification number. For instance, *03* is defined for the *Loopcount(int, Mode_name)* annotation. Based on the offset of the method of the JBC in the JCF file, the annotation of the *Loopcount(int, Mode_name)* can be given as *<03 #10, 10, Quick_Mode>* and *<03, #10, 30, Normal_Mode>* in the body area.

```
...
    <!-- WCET information -->
    <TAG=1>
    <WCET>
          <SPEC>
              <SUBTAG_COUNT=4>
              <Method=Constant_Pool_index>
              <Method_Body>
                    <00 PC_Offset=int, Mode=name>
                    <01 PC_Offset=int, Use_Mode=name>
                    <02 PC_Offset=int, LoopCount=int>
                    <03 PC_Offset=int, LoopCount=int, Mode=name>
              </Method_Body>
          </SPEC>
          <TAG_BODY>
              <!--Constant_Pool index #36=
                    SimpleIO$ControllerThread.Call_ForLoop(I)V -->
              <Method=#36>
              <Method_Body>
                    <00 #1, Quick_Mode>
                    <00 #1, Normal_Mode>
                    <02 #10, 50>
                    <03 #10, 10, Quick_Mode>
                    <03 #10, 30, Normal_Mode>
              </Method_Body>
              ...
          </TAG_BODY>
    </WCET>
...
```

Figure 4. A fragment of the XAC file for Call_ForLoop method

6. Integrating with Real-Time Specification

This section is mainly concerned with how to extract real-time thread parameters from JCF files. In the XRTJ architecture, the XRTJ-Analyser has an interface to support different specifications [4,10] of the real-time extensions to Java. Developers can define which specification is going to be used in the system by the system configuration file. To do a schedulability analysis, real-time thread parameters are required. An example using RTSJ is given in this section.

In the RTSJ [4], a real-time thread (*javax.realtime.-RealtimeThread*) is an extension of the primary Java thread (*java.lang.Thread*). A real-time thread has a scheduling parameters object (*javax.realtime.SchedulingParameters*) that contains the priority of the thread. The object may also provide other parameters, such as *importance* value, for a particular scheduling algorithm. Each real-time thread is associated with a dispatching parameters object *javax.realtime.ReleaseParameters*) that includes cost, deadline, and two asynchronous event handlers. In the RTSJ, periodic threads, aperiodic threads, and sporadic threads are classified by the characteristics of their dispatching parameters. These objects extend from the object *ReleaseParameters,* such as *PeriodicParameters* object, *AperiodicParameters* object, and *SporadicParameters* object.

The XRTJ-Analyser reads JCF files and finds out the parameters objects associated with the specific real-time threads as follows. First of all, the XAC-Analyser reads the real-time thread information from the constant pool table stored in the JCF files. Solving the symbolic information, the full name and type of each object can be reproduced. In line with the chosen real-time specification interface, the analyser, using the complete information about these objects and code attributes, can easily find out each real-time thread created in the program. Following this, the relationships between parameter objects and the real-time threads are identified. Finally, all real-time thread parameters are collected.

```
...
0         new #46 <Class javax/realtime/PriorityParameters>
3         dup
4         bipush    8
6         invokenonvirtual #49
              <Method javax/realtime/PriorityParameters.<init> (I)V>
9         astore_1
...
118       new #68 <Class javax/realtime/PeriodicParameters>
121       dup
122       aload     4
124       aload     5
126       aload     8
128       aload     5
130       aconst_null
131       aconst_null
132       invokenonvirtual #71
              <Method javax/realtime/PeriodicParameters.<init>
                    (Ljavax/realtime/HighResolutionTime;
                     Ljavax/realtime/RelativeTime;
                     Ljavax/realtime/RelativeTime;
                     Ljavax/realtime/RelativeTime;
                     Ljavax/realtime/AsyncEventHandler;
                     Ljavax/realtime/AsyncEventHandler;)V>
135       astore    11
...
175       new #73 <Class SimpleIO$ControllerThread>
178       dup
179       aload_1
180       aload     11
182       invokenonvirtual #76 <Method SimpleIO$ControllerThread.<init>
                    (Ljavax/realtime/SchedulingParameters;
                     Ljavax/realtime/ReleaseParameters;)V>
185       astore    14
...
219       aload     14
221       invokevirtual #84 <Method java/lang/Thread.start ()V>
...
```

Figure 5. A fragment example of the Java Byte Code

An example fragment of a JCF file is given in figure 5. The context in which this fragment is used creates a periodic thread (*SimpleIO$ControllerThread*) in a main program. In the figure, we can see that the periodic thread, which is created in offset byte 175-185, contains a scheduling parameters object and dispatching parameters object, which are created in offset byte 0-9 and offset byte 118-135 respectively. Analysing these Java byte codes, the priority of the periodic thread and its dispatching parameters, including *period, deadline, cost* and *start* values, can be extracted easily. The complete example is given on our website (http://www.xrtj.org).

In a very similar way, real-time thread parameters from the system using RT-Core specification [10] can be accumulated.

7. XRTJ-Analyser

As mentioned in previous sections (Sec.5 and Sec. 6) real-time parameters, including priority and dispatching parameters, for the set of threads and WCET estimates can be produced from the JCF and XAC files. Given the WCET estimates and real-time parameters, the schedulability analysis can be conducted easily. In the XRTJ-Analyser, only the system configuration information is needed. The system configuration includes: which scheduling algorithm is going to be used in the system, such as *Earliest Deadline First*; which resource access protocol is applied in the scheduling algorithm, such as *priority inheritance protocol*; which real-time specification is being used, such as RTSJ [4]; and additional information about VMTM, such as memory management features. In addition, the scheduling algorithms can be provided through the interface which is offered by the XRTJ-Analyser to support various scheduling algorithms. This feature can be easily achieved as a result of the way in which the dynamic binding and dynamic loading of object-oriented features are applied.

Following the system configuration, the XRTJ-Analyser loads the scheduling algorithm and carries out the schedulability analysis. Scheduling algorithms must provide scheduling characteristics, algorithms which can calculate other scheduling parameters, such as *release-jitter, blocking time, response-time*, and resource access protocols which are provided to manage the priority inversion problems. Further details of the schedulability analysis are not discussed here since they are outside the scope of this paper. Finally, the XRTJ-Analyser produces the result of the analysis of the system. The output file provides not only the result of the analysis, but also includes timing and scheduling information, such as *response time, release-jitter, blocking time.*

8. Current Status and Future Work

Currently, we are developing a translator to produce XAC files from the annotated Java programs. At the same time, we are working on the schedulability analyzer to integrate it with the XRTJ-Analyser.

In this paper, we have only mentioned the static timing analysis issue. Indeed, the approach can be applied in the run-time environment to validate the timing constraints of the system dynamically. To perform this, the Java virtual machine needs to be modified. Our future work includes:

- developing the XRT-JVM (Extended Real-Time Java Virtual Machine) to support dynamic timing analysis at run-time
- developing the XRTJ-Compiler to produce the XAC file and JCF file during compiling time
- extending the portable WCET analysis to get tighter estimated values.

The most updated information can be found on our website (http://www.xrtj.org).

9. Conclusions

Since the aim of portable code is to support hardware interchangeability, the validation of the timing constraints in a particular run-time system is of vital importance for hard real-time systems. We have presented a static timing analysis approach based on the Java architecture to analyse the execution time of hard real-time systems during the development phase. The approach demonstrates the expressive power of the XRTJ architecture in terms of the static timing analysis in a portable code context. This architecture can be easily merged with other approaches, such as model checking.

10. Acknowledgements

The authors would like to thank Peter Puschner and an anonymous reviewer for their comments on the early draft of this paper.

11. References

[1] 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)*, pp.39-48, South Korea, December 2000

[2] 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

[3] 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

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

[5] E. Briot. JGNAT: The GNAT Ada 95 environment for the JVM. In Ada France 1999

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

[8] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. Technical Note 2000-003, *Compaq Systems Research Center*, 2000.

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

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

[11] G. T. Leavens, A. L. Baker and C. Ruby. JML: A notation for detailed design. from H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175-188. Kluwer Academic Publishers, Boston, 1999.

[12] 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

[13] ObjectAda, Anoix (http://www.anoix.com)

[14] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs, *Real-Time Systems Journal*, 1(2): pp.159-176, September 1989

[15] P. Puschner and A. Schedl. Computing maximum task execution times - a graph based approach. *Real-Time Systems*. 13(1): 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*, pp. 15-22, *ISORC* 2001

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