Analyzing resource conflicts in Java programs

Report

presented in June 25th, 2014

in order to obtain

Master de l'Université de Lorraine

(Logiciels: méthodes formelles et ingénierie)

by

Jorge Ibarra Delgado

Jury composition

Didier Galmiche
Stephan Merz
Marie Duflot-Kremer
Horatiu Cirstea

Laboratoire Lorrain de Recherche en Informatique et ses Applications

Acknowledgments

I would like to thank my supervisors Stephan Merz and Marie Duflot-Kremer who encouraged and guided me through my internship and were very patient and respectful. I am sure I have made the right choice working with them.

I also want to thank the people in the team, specially Haniel, Jingshu and Federico. I hope the problems of getting a table for all the team at lunch time will be solved soon!

My sincere thanks to the rest of the committee: prof. Galmiche and prof. Cirstea for their encouragement, insightful comments, and hard questions.

Last but not least, I would like to thank my family at Mexico and all of my friends I've met this year who where with me on good and bad times.

Contents

Chapter 1

Introduction

1.1 Overview

In the course of the last years, with the improvement of technology, the development of concurrent complex systems became a common practice.

But what is a concurrent system? As an informal answer, we can see them as hardware or software systems interacting (or not) with each other. These systems perform diverse processes where it is possible to compete for access to resources they are sharing. Therefore it is essential to have some mechanism to coordinate these processes, e.g., in a software system, a piece of data is located in a shared memory location and a scheduler is in charge to assign turns to the different threads willing to access to the critical section.

Now, many of this concurrent systems can be considered also as real-time systems. These are systems where an action is executed according to an external input stimuli and this action is expected to be performed in a specified amount of time, we expect these systems to be correct when the action is done in time but also when the results are correct.

We can find these kind of systems all around the world in our everyday life. We can consider for instance an on-line multi-player game, where we expect our actions will be reflected at the client of our friends and adversaries. Another example is the system that make the landing gear of an aircraft to go down while the pilots of the aircraft are maneuvering a landing action. Finally we can consider an airline reservation system where we have to process the request from clients confirming their bookings and checking-in, sometimes printing the boarding passes from the airline counter 5 minutes before closing the gate. We can name also other examples like a pacemaker, a driver-less car, the stock exchange system or even YouTube.

Real-time systems can be classified according to certain required characteristics, this features are the importance of meeting the deadline on time and the obtained result.

- Soft real-time systems
- Firm real-time systems
- Hard real-time systems

In short words, a soft real-time system is one that can manage to keep executing and the results are valid even though failures present; firm real-time systems also can manage to present some failures, but if that happens the system can bring trivial results, and sometimes, if the failures persist, a catastrophic failure may occur. Hard real-time systems, which are also called critical-safety real-time systems are characterized by not tolerating failures, if a single failure occurs it may lead the system to a irreversible or irreparable state.

Retaking the examples mentioned before, we can consider the multi-player on-line game as a soft real-time system: if there are some glitches in the network the system may show some lagging and data loss, but in most of the cases the game session can still be played, with only some exigent players being angry. The reservation system can be taken as a soft real-time system or even a firm real-time system, it depends on which kind of failure and consequence we can focus on. If a client makes the reservation of a plane successfully but the data administration fails and in reality the plane is already full, he will get anyway his ticket. On the other hand, giving the wrong data about departures can make a customer to loose his plane, giving a not wanted result. For the aircraft case it's quite easy to determine that if some of the software/hardware mechanisms are not working we may have a very serious consequence that can cost several human lives.

Because of the nature of concurrent systems, developers now have not to think only in a sequential programming manner, but to add to their knowledge factors like data races, starvation, deadlocks and scheduling.

Failures due to timing issues are still a hot topic. Detection by common quality assurance methods, like testing, are still a difficult task. For instance, in a sequential program we can determine an error by giving some set of inputs and initial state and then track the bug, which is not an easy task in a million lines code, but it is feasible. However with concurrent programs the results are affected by a such high quantity of factors that finding the cause of a bug is a very difficult and time consuming work.

In safety-critical real-time software systems, bad synchronization may lead to data corruption.

1.2 The project and contributions

Having the interest to research the aforementioned subject, The AJITPROP (Analysis of Java Timing Properties) project was created, partially founded by the Airbus Group Corporate Foundation[1]. The project is a collaborative effort involving members of the Veridis team[2] at LORIA / Inria Nancy and the Acadie team[3] at IRIT / Université de Toulouse. Our task is to propose methods to solve the detection of resource conflicts in multi-threaded Java programs statically (i.e., prior to execution).

The analysis will be carried out by abstracting the source code of Java programs to appropriate logical structures, namely timed automata or mixed arithmetic-Boolean formulas. As an intermediate step, the translation from source code to a logical structure will be certified by using a proof assistant, in our case Isabelle [11]. Then finally the possible conflicting access to the resources will be detected by using techniques of formal verification, particularly model checkers like Uppaal[2] or SMT (Satisfiability Modulo Theory) solvers like Z3[4] or veriT[5].

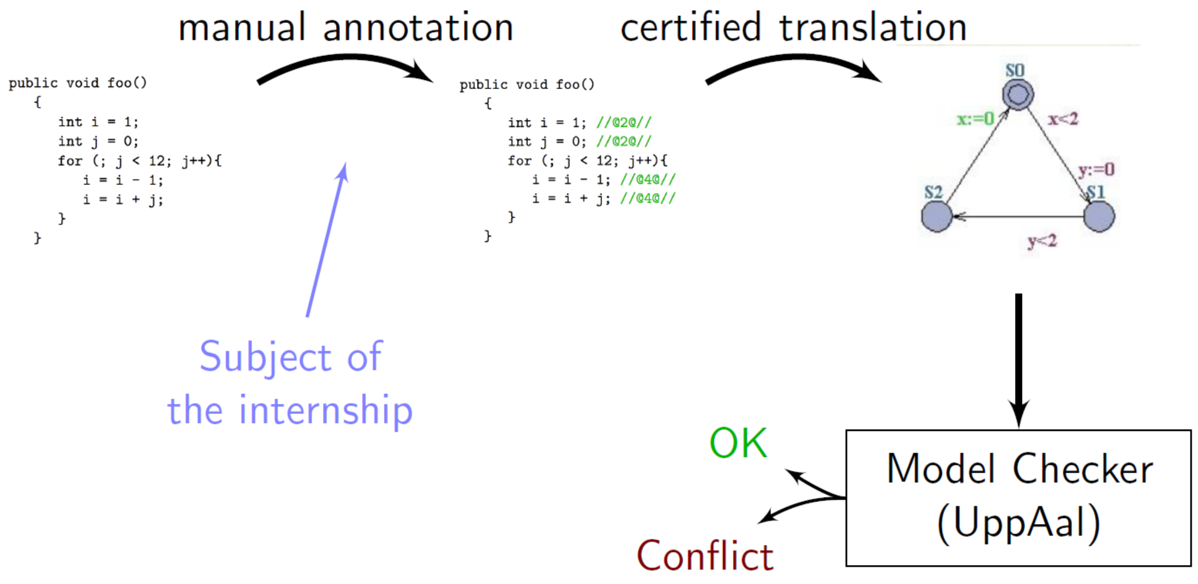
In order to start the analysis, we assumed that Java source code was being annotated with worst-case timing information indicating the execution time of each line of code. Then this annotated code could be translated to timed automata as mentioned before.

These annotations where provided manually, by rule of thumb. That signified a very possible miscalculation and inaccurate results were suspected. Not to mention the time consumption it takes to calculate by hand each chunk/line of code.

The objective of my internship was to create a program which could use as input a Java source code without any timing information, analyze the code by calculating the timing of each statement and then, as an output, automatically provide annotations of the execution time to the code. The map of the entire project can be seen in Figure 1.1

As a first step, a proposed and possible immediate solution was to get familiarized with JOP[16] (the Java Optimized Processor) to provide the desired results to the team. Unfortunately, the features of JOP's WCA (Worst Case Execution Time Analyzer) tool[19] do not comply with what was required. Hence, as a second step, using what was learned from WCA and its APIs we created the Execution Time Analyzer tool prototype, which provides what the team needs: automatic annotation of execution timing for each line of code in Java source code.

[1]`http://fondation.airbus-group.com`
[2]`http://veridis.loria.fr`
[3]`http://www.irit.fr/-equipe-acadie-`



```
public void foo()
   {
      int i = 1;
      int j = 0;
      for (; j < 12; j++){
         i = i - 1;
         i = i + j;
      }
   }
```

```
public void foo()
   {
      int i = 1; //@2@//
      int j = 0; //@2@//
      for (; j < 12; j++){
         i = i - 1; //@4@//
         i = i + j; //@4@//
      }
   }
```

Figure 1.1: The work-flow of the Analysis of Java Timing Properties project. First we annotate the Java code with timing information, then the code gets translated to timed automata and finally it is checked by UppAal. The focus of my internship is to provide an automatic solution to the manual annotation.

1.3 Outline of this work

Chapter 2 provides information about the context of real-time systems in a formal manner, emphasizing embedded and critical systems and introduces some basic notions of the concept WCET (Worst Case Execution Time). Then we introduce the problems of resource conflicts in real-time programing and a possible solution, mentioned partially here.

In chapter 3 the subject about WCET solving will be the pilar of this part. We will talk of the features of the Java Optimized Processor and its WCET analyzer tool WCA. It will provide some of the observations made during testing and the conclusions that derived into the development of the Execution Time Analyzer. Finally, information about other WCET solvers will be provided.

The Execution Time Analyzer tool is presented in chapter 4. We will expose the details we recovered from WCA in order to reuse them in our tool. Afterwards we will present the execution stages of the tool with examples and the difficulties found in its development.

Finally, chapter 5 discuss some of the further works to be done and concludes this work.

Chapter 2

Backgrounds and project

2.1 Real time systems

To have a better understanding of the contents of next chapters, it is pertinent to get into the context of real time systems.

In [8] a real time system can be defined with any of these both definitions:

The first one, seen more as a practical definition, states that a real-time system is a computer system that must satisfy bounded response-time constraints or risk severe consequences, including failure. Considering failure as an unsatisfiability of the requirements indicated in the system requirements specification. The second definition states that a real-time system is one whose logical correctness is based on both the correctness of the outputs and their timeliness.

In real-time systems, is very common to have events happening all the time. An event [8] occurs when there is a change in the flow-of-control of the sequence of a program. E.g., case, if - then, and while statements in any programming language represent a possible change in flow-of-control. In object-oriented languages instantiation of an object or the invocation of a method causes the change in sequential flow-of-control.

There can be either synchronous or asynchronous events:

- Synchronous events occur when there's a time predictable occurrence in the flow-of-control, such as the events done by a conditional branch or by an internal interruption. We can anticipate this.
- Asynchronous events occur at unpredictable points in the flow-of-control and are caused by external sources. This can be happening for instance when a user is clicking the mouse button, the program then gets the information as an input and reacts accordingly to the specified time and requirements.

Practically almost all real-time systems are reactive or embedded systems:

- Reactive systems are those in which task scheduling is driven by ongoing interaction with their environment. An example of this is a system that executes an action when the user pushes a button.
- An embedded system contains one or more computers having a central role in the functionality of the system, but the system is not explicitly called a computer. A common instance to understand this is an automobile: we can find many embedded processors that control the airbags, brakes, air conditioning, fuel injection and so on.

In most of the subject's literature we can recognize mainly three types of real-time systems according to their tolerance to failures:

- Soft Real-Time Systems: those systems whose performance can be affected negatively but keep working and computations results are still usable if a failure to meet timing constraints is presented.

- Firm Real-Time Systems: we can recognize a firm real-time system if the system can withstand some failures meeting timing constraints but the obtained computations are futile. [7]
- Hard Real-Time Systems: Also known as safety-critical real-time systems, are applications where the outcome of not meeting a single deadline may lead to complete or catastrophic system failure.

As an example to illustrate the different types of real-time systems, we can look at the following table.

System	Real-Time Classification	Explanation
Avionics weapons delivery system in which pressing a button launches an air - to - air missile	Hard	Missing the deadline to launch the missile within a specified time after pressing the button may cause the target to be missed, which will result in catastrophe
Pacemaker		Missing the deadline to deliver the electrical pulses to the muscles may cause a heart stroke and death for the patient
Navigation controller for an autonomous weed - killer robot	Firm	Missing a few navigation deadlines causes the robot to veer out from a planned path and damage some crops
Prototype driverless car		Missing some of the deadlines provokes the car to steer or stop in unappropriated manner, probably causing some damages to the car and the test environment
Console hockey game	Soft	Missing even several deadlines will only degrade performance
Online video streaming		Missing deadlines will only cause the video to stop

Table 2.1: Samples for Hard, Firm And Soft real-time systems, some of them taken from [8, p.7]

Thiele and Wilhelm [21] define the following key concepts in a clear form:

- Interference: If there is a dependency between the execution time and some non-observed external behavior, then we will say that the time interval is nondeterministic with respect to the available information. Therefore, there will be a difference between the worst case and the best case behavior. If the upper and lower bounds are computed (or measured) using the same limited knowledge about the whole system, then we clearly can not achieve a smaller interval between the upper and lower bounds.
- Predictability: The time predictability of a system is related to the difference between upper and lower bound on execution time. A low predictability (large difference between upper and lower bound) is caused by interference and limited analyzability.
- Performance: The worst case and best case performance are related to the worst case and best case execution time, respectively. The average case performance measures the average execution time. Threats to a high worst case performance are usually caused by interference from unavailable or unknown information about the system or its environment.
- Guarantee: The worst case and best case guarantee are linked to the upper and lower bound on the execution time. Small worst case guarantees may be caused by interference and limited analyzability.

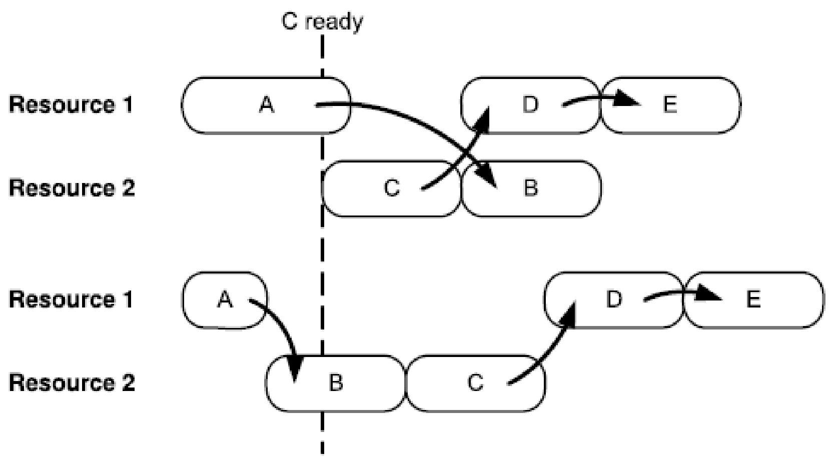


Figure 2.1: Scheduling anomaly, taken from [14].

Also, there's another term defined in [1] that is appropriate to mention:

- Timing Anomalies[10]: A timing anomaly is a situation where the local worst case does not contribute to the global worst case. For instance, a cache miss may result in a globally shorter execution time than a cache hit because of scheduling effects, cf. Figure 2.1 for instance. Shortening instruction A leads to a longer overall schedule, because instruction B can now block the more important instruction C. Analogously, there are cases where a shortening of an instruction leads to an even greater shortening of the overall schedule.

If we pay attention to the last definition, it looks like such an effect can happen only if we have more than one active component, or, in our project's context, a processor. However, in the scope of our project we will consider scheduling on single processors.

2.2 Solving resource conflicts

An important problem to deal with in concurrent real-time systems is to avoid conflicting accesses to shared resources, which we can find also in embedded systems. The most current solution to this problem is to use locks, but unfortunately, this solution carries well-known issues, such as run-time overhead associated with acquiring locks and recurring errors and deadlocks. Also, using locks may be incompatible with the strict requirements on the predictability of execution time. A solution that is being studied within the AJITPROP project is to rely on the constraints on the scheduling of parallel threads of real-time systems in order to ensure that no conflicting accesses occur.

To do so, we first take a plain Java source code and annotate it with timing information, the next example illustrates what do we want to do:

Listing 2.1: Two sets of code that will be executed concurrently by two or more threads

```
//Thread 1
a = a + 1;
b = a;
```

```
//Thread 2
b = a;
c = a − b;
```

Listing 2.1 is a simple code depicting two threads updating the values a, b and c. After executing the end of the program, we want c to be 0, meaning that a and b will be equal. Keeping the code as it is doesn't guarantee our requirement because we don't know at which time the threads are going to access to the resources.

Listing 2.2: The chunks of code now are being synchronized in order to avoid a resource conflict

```
//Thread 1
synchronized(a){
  a = a + 1;
  b = a;
}
```

```
//Thread 2
b = a;
synchronized(a){
  c = a − b;
}
```

A possible solution to control thread synchronization is by using locks, as can be shown in listing 2.2. With that, the threads can only have access one by one to the critical section. Nonetheless, synchronizing is not the best option because of the cost of resources taken at each invocation of the synchronize method and the possibility of having errors like deadlock.

Listing 2.3: Instead of synchronizing, the code is analyzed according the sleep statements and the annotations on code

```
//Thread 1
sleep(1);
//@2@//
a = a + 1;
//@1@//
b = a;
```

```
//Thread 2
//@1@//
b = a;
sleep(3);
//@2@//
c = a − b;
```

For programs written for real-time execution platforms where all threads share a common global time reference, such as Safety-Critical Java [20], another way to solve the synchronization of the threads is by scheduling constraints. Looking at the Listing 2.3 we can see that every line of code, except of the sleep statements, are annotated with execution time. We are assuming that the code will be processed using a particular processor architecture like the one in the Java Optimized Processor. We also assume that threads are scheduled for execution on a single processor, managed by an eager scheduling policy where some thread executes whenever at least one thread is executable.

So, in the example mentioned at Listing 2.3 the first thread sleeps while the second one is scheduled to execute the first line of code for one time unit. After that, the second thread is put to sleep for the period of time indicated at the code of the first thread (3 time units). Then Thread 1 is scheduled and can execute the two lines of code. After this point, the second thread is able to execute it's code, giving us the wanted output. However, as we can notice, one must be careful handling the timing annotations in order to obtain the desired result.

After this, the program will be translated to a formal structure, like timed automata or quantifier-free linear integer arithmetic. In the case of making the translation into a timed automata, the model is then verified by uppaal, which is integrated tool environment for modeling, verification and validation of real-time systems in the form of timed automata. As an alternative to the prior method, the translation can be feed to an SMT solver, which can efficiently decide if the formula is satisfiable or not, giving a counter model if something in the formula goes wrong.

So far we have seen some concepts of real-time systems and concurrency, also we exposed how we are trying to solve the resource conflict in concurrent real-time systems. The next chapter will explain the key ideas to obtain the correct scheduling time, based on the concept of getting the WCET (Worst Case Execution Time) of a program and its relation with the Worst Case Analyzer from JOP.

Chapter 3

Solving the WCET

3.1 WCET

Because of their high necessity to be fail-safe, hard real-time systems are the main scope of my research's subject. In order to analyze them, it is important to know the computation or execution time that usually depends to some extent on the input data and other variable conditions [23]. It is then important to find the WCET (Worst Case Execution Time) and to verify that it's short enough to comply with the required timing deadlines.

Also, as a consequence of finding the WCET, another important point to take care of is [25] that we need to restrict the programming style: recursion is not permitted as the iteration counting of loop structures must be provided. It is important to point that, there are ways to provide an upper bound on the number of cycles a loop has to perform in a WCET analyzer if the counting is unknown, like the case when a loop is modifying the counter. An example will be provided in 3.2.2. A reliable guarantee based on the worst-case execution time of a task could easily be given if the worst-case input for the task were known. Unfortunately, in general the worst-case input is not known and hard to obtain.

In figure 3.1 are explained many concepts of a real-time task. Execution time varies according to input given and other environment variables. The shortest execution is called the best case execution time and the longest execution is called the worst case execution time. The rest of the figure is represented by a lower curve, which represents a subset of measured executions. Its minimum and maximum are the minimal observed execution times and maximal observed execution times, resp. The darker curve, an envelope of the former, represents the times of all executions. Its minimum and maximum are the best-case and worst-case execution times, respectively, abbreviated BCET and WCET. As one can imagine, exploring all the possibilities in order to establish an exact best-bound and worst-bound could be very a extensive task, if not impossible.

In industry, one of the most used techniques to estimate execution time is the measurement of point to point execution time of the action to perform for a subset of possible test inputs. This will determine the minimal observed and maximal observed execution times, which, generally, overestimate the BCET and underestimate the WCET, making this method not safe for safety-critical real-time systems. Most of the time, this method is called dynamic timing analysis.

Computation of the bounds on the execution time of a program can be feasible only by considering all possible execution times, i.e., using all the possible ways to execute the task. This is done by abstracting the task to make timing analysis possible. The code, being abstracted to another form like a control flow graph or timed automata, is not being executed, hence, it is analyzed statically. However, abstraction implies a loss of information, then the calculated worst case execution time bound usually overestimates the exact WCET and vice versa for the best case execution time. The obtained WCET bound is a representation of the worst-case timing a method can deliver.

So, taking the pros and cons, [24] the static analysis approach computes safe upper bounds of the WCETs of program fragments, and thus also of tasks. Nonetheless, abstracting hardware models tends to produce errors and it takes time to construct the models, especially if there's no precise specifica-

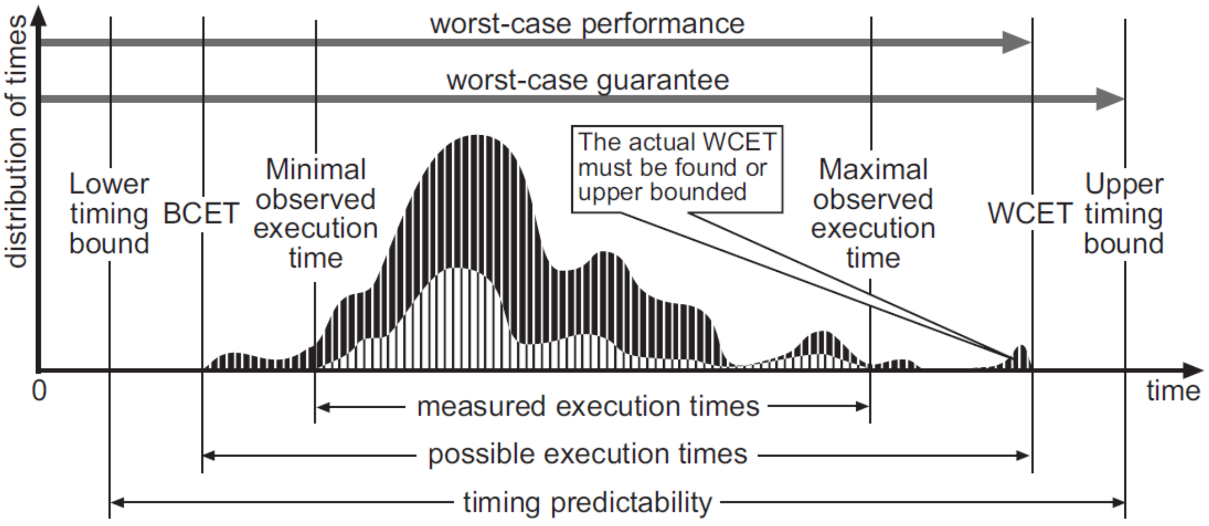


Figure 3.1: Basic notions concerning timing analysis of systems, taken from [25]

tion available. Using measurement is more easily portable to new architectures, as it does not rely on abstraction of models. But, soundness over the results of measurement-based approaches is difficult to guarantee. Of course it could be trivially sound if all initial states and inputs would be covered. However we can imagine that this is practically not feasible. Only a subset of the initial states and inputs can be considered in the measurements.

The precision of the mentioned methods depends on the procedures used for timing analysis and on general system properties, like software characteristics or hardware architecture. We have to take account also these properties can be contained under the notion of timing predictability.

We would like to point out that the Worst Case Analyzer, which is the tool that we studied, belongs to the static analysis classification, in the next section we will explain the characteristics that makes it part of that category.

WCA is based on two ideas to calculate the WCET.

The first one, introduced by Puschner and Schedl [12], calculates the WCET by transforming it to an integer linear programming (ILP) problem. Having a timing graph, each basic block is represented by an edge e_i with the weight of the execution time of the basic block. Vertices v_i represent the split and join points in the control flow. Moreover, each edge is also assigned an execution frequency f_i. The constraints resulting from the timing graph and additional functional constraints, like loop bounds, are solved by an ILP solver.

The second approach, published by Li and Malik [9] follow a similar approach with ILP. However, instead of using a timing graph, they use a control flow graph as the basis to build the ILP problem.

Further in this document we will explain how these two ideas where used to calculate the WCET in WCA.

3.2 JOP and WCA

3.2.1 JOP

JOP is an implementation of the Java Virtual Machine (JVM) in hardware, it is made to be used mainly on Field Programmable Gate Arrays (FPGA) like the Cyclone series from Altera. Also, its design is focused from the ground up with time-predictable execution of Java bytecode as a major design goal [17]. There are many factors that will be explained in this section that make JOP a good candidate to be a time predictable processor: the translation of Java bytecode intro microcode; the set of stack instructions;

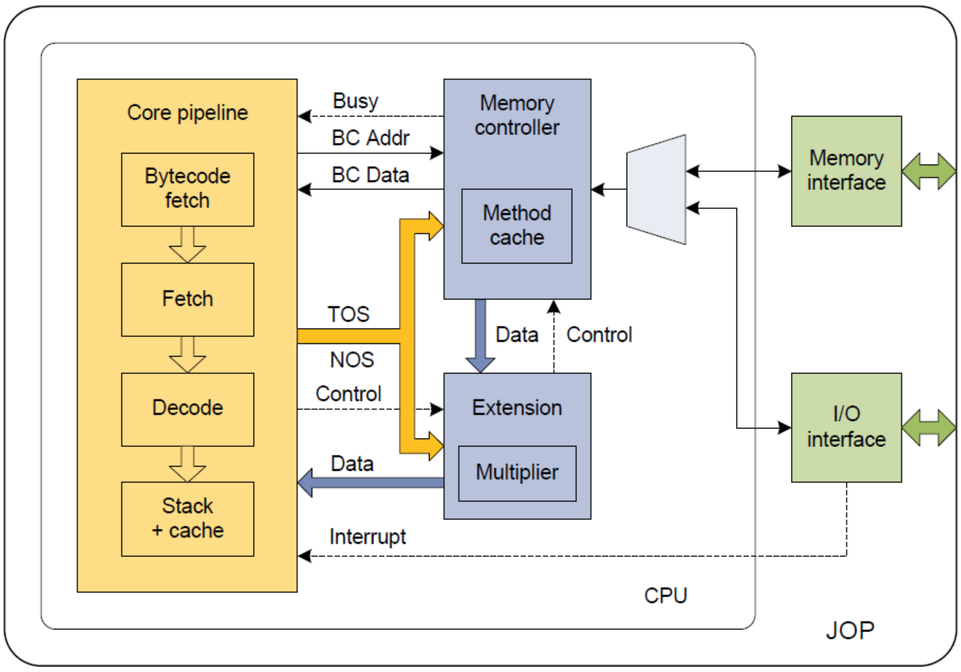


Figure 3.2: Block diagram of the Java processor JOP, taken from [19].

the real-time stack cache and the method cache. JOP is the smallest and fastest Java processor available today [15].

The main difference we can remark between the Java Virtual Machine and JOP is their stack architecture. JVM is a CISC stack architecture, whereas JOP is a RISC stack architecture.

Figure 3.2 shows JOP's major functional units. As we can see, a common configuration of JOP contains the processor core, a memory controller, a memory interface, a extension module and a number of I/O devices. The processor contains four stages, from which three are the microcode pipeline stages: microcode fetch, decode and execute; the remaining stage is an additional translation step called bytecode fetch. The core reads bytecode instructions through dedicated buses: the BC address and BC data, from the memory controller. The extension module provides the link between the processor core, and the memory and I/O interfaces, it also controls data read and write. The main memory and I/O devices are connected via the memory controller, which also contains the method cache. The busy signal is used by a microcode instruction to synchronize the processor core with the memory unit. The execution of microcode is done concurrently to memory access. TOS and NOS means top of the stack and next on stack respectively, these are the ports to the other modules.

There are no processor resources being shared throughout bytecode boundaries. As a result, JOP does not generate pipeline dependencies between two bytecodes that could produce an unbounded timing effect. Hence, the processor is designed to avoid any timing anomalies, which where mentioned in chapter 2 and are commonly found in standard microprocessors [10].

There are two instruction sets that need to be discussed: bytecode and microcode. Bytecode is the set of instructions of a Java compiled code and it is executed by the Java Virtual Machine. It consists of 256 instructions. Microcode is the exclusive set of instructions of JOP. At the moment of execution, bytecode instructions are translated into microcode. JOP implements 54 different microcode instructions. Some of the bytecode instructions are translated to a single microcode instruction. There are other more complex bytecode instructions that require to be translated to a sequence of microcode instructions. Also it is important to mention that some of the bytecode instructions like *new*, which creates and initializes a new object, are very complicated to implement in hardware, then these instructions have to be emulated by software. In JOP this emulation is done by using Java code, meaning that, in order to implement a complex bytecode, the developers of JOP re-created the instruction using other bytecodes. This will be observed when we describe some of the observations when executing WCA. At the time [17] was published, 43 of the 201 different bytecodes are implemented by a single microcode instruction, 92 by a microcode sequence, and 41 bytecodes are implemented in Java. Furthermore, JOP contains additional bytecodes that are used to implement low-level operations, such as direct memory access.

At the cache level, standard caches are difficult for WCET predictability. For that reason JOP proposes two types of cache which are time predictable: a stack cache as a substitution for the data cache and a method cache to cache the instructions.

- Stack cache: it is the stack containing method local variables, used for the stack operations is a heavily accessed memory region. Therefore the JOP stack cache is organized in two levels. The first level are two discrete registers for the top of the stack and the Next on stack. Then the second level is placed in on-chip memory. This stack cache is not exchanged with the main memory automatically because it would be difficult to analyze. Instead, the exchange can be done at method invocation or return, or also at a thread switch.

- Method cache: A Java application typically is constructed by short methods. Also, in [16] is shown that the largest method in the Java runtime library (JDK 1.4) is of 16706 bytes. There are no branches out of the method and all branches inside are relative. Having in mind this factors, the architecture proposed consists of loading the full code of a method into the instruction cache prior to execution. The cache is filled on invocations and returns. That means that the cache miss penalty is known and this instructions are put together. The other instructions are cache hits. As the cache store whole methods, it is named method cache.

  The main difference between the method cache and a conventional cache is that the blocks for a method are all loaded at once, and need to be consecutive [19]. This constraint has as a consequence the restriction of replacement policies. In WCET related literature [13, 1] Least Recently Used (LRU) is one of the best analyzable methods, however it is not possible to use within JOP's architecture. Instead, it uses a First In First Out (FIFO) based replacement policy.

3.2.2 WCA

As briefly mentioned before, in order to make the calculation of the WCET a decidable problem, it is necessary to impose some program restrictions:

- Programs must not contain any recursion. Recursive algorithms have to be transformed to iterative ones. There is no support for tail-recursive calls in microcode because it is suggested not use recursion in embedded systems with restricted memory.

- Absence of function pointers. In the case of Java, having function pointers is unavoidable because function pointers are very similar to inherited or overriden methods, which are dispatched at runtime. In contrast to other language's function pointers, in Java it is possible to statically analyze which methods can be invoked when the whole program is known. In [18] this restriction is substituted by the following one:

- Dynamically class loading is not allowed. We need to know beforehand which classes are being used in the program to do the static analysis, therefore loading them dynamically is of little use.

- The upper bound in the number of iterations of each loop has to be known.

WCA is an open source Java command-line tool used to calculate the worst case execution time. At the moment, as many of the current WCET solving tools, it is used as it is only in its target processor, which is JOP. However, in [17] it is stated that with some modifications, there is a possibility that the tool can be used for other processors, like the real-time Java processor jamuth [22].

```java
public static int loop(boolean b, int val) {
    for (int i = 0; i < 10; ++i) { // @WCA loop=10
      if (b) {
        for (int j = 0; j < 3; ++j) { // @WCA loop=3
          val *= val;
        }
        return val;
      } else {
        for (int j = 0; j < 7; ++j) { // @WCA loop=7
          val += val;
        }
        return val;
      }
    }
    return val;
  }
```

Listing 3.1: A java code example

WCA calculates the WCET based on the implicit path enumeration technique approach, this means that the WCET is computed by identifying which path is more costly in a directed graph that represents the control flow of the program.

The tool performs the following steps to calculate the WCET and bring the results, along with these points we will include an example to clearly illustrate what happens at each step, the code to analyze is the one located in Listing 3.1.

1. Extraction of the basic blocks. In the context of bytecode, a basic block is a sequence of instructions without any jumps, except as last instruction, or jump targets within this sequence [19]. The analysis of the WCET is made at the bytecode level. Thanks to the knowledge of the bytecode's execution time previously mentioned in 3.2.1 the calculation task is simplified. As there are no pipeline dependencies, the calculation of the execution time for a basic block is just adding the individual cycles for each instruction. To access the class files, the Byte Code Engineering Library (BCEL) [6] is used. BCEL is a toolkit that allows static analysis and dynamic modification of Java class files and bytecode of the methods.

2. Building of the Control Flow Graph (CFG). After extracting the basic blocks using the BCEL. The CFG is built as shown in Figure 3.3.

3. Detection of loops. To calculate properly the WCET, WCA must detect the loop structures, like while and for. Normally, the WCA is capable of detect the number of cycles to be executed in each loop but only if the conditional part of the loop is simple and if the method to analyze is part of the execution. If this conditions are not met, it is possible to put annotations by the user in the code indicating the number of cycles each loop has to perform. These annotations can be found clearly in Listing 3.1

4. Solve the WCET. WCA uses the open-source ILP solver [3] which is a Mixed Integer Linear Programming (MILP) that is integrated into the WCA by invoking it via the Java library binding.

   As explained in [17], in the CFG, each vertex represents a basic block B_i with execution time c_i. With the basic block execution frequency e_i the WCET is:

$$WCET = max \sum_{i=1}^{N} c_i e_i$$

   The maximum value of this expression results in the WCET of the program. Also, each edge is assigned an execution frequency *f*. These execution frequencies represent the control flow through the WCET path and are the integer variables that are being calculated in order to solve the ILP problem. There are two primary constraints to form the ILP problem:

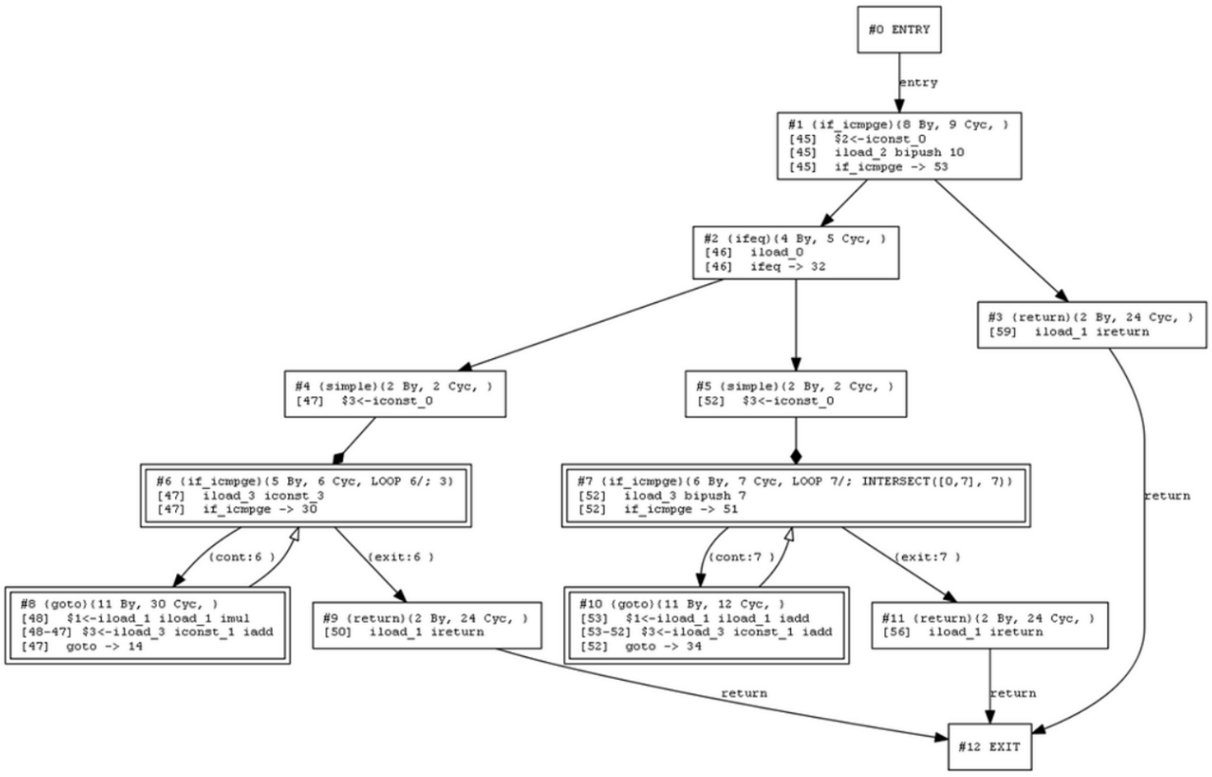

#0 ENTRY
entry
#1 (if_icmpge)(8 By, 9 Cyc,)
[45] $2<-iconst_0
[45] iload_2 bipush 10
[45] if_icmpge -> 53
#2 (ifeq)(4 By, 5 Cyc,)
[46] iload_0
[46] ifeq -> 32
#3 (return)(2 By, 24 Cyc,)
[59] iload_1 ireturn
#4 (simple)(2 By, 2 Cyc,)
[47] $3<-iconst_0
#5 (simple)(2 By, 2 Cyc,)
[52] $3<-iconst_0
#6 (if_icmpge)(5 By, 6 Cyc, LOOP 6/; 3)
[47] iload_3 iconst_3
[47] if_icmpge -> 30
#7 (if_icmpge)(6 By, 7 Cyc, LOOP 7/; INTERSECT([0,7], 7))
[52] iload_3 bipush 7
[52] if_icmpge -> 51
return
(cont:6)
(exit:6)
(cont:7)
(exit:7)
#8 (goto)(11 By, 30 Cyc,)
[48] $1<-iload_1 iload_1 imul
[48-47] $3<-iload_3 iconst_1 iadd
[47] goto -> 14
#9 (return)(2 By, 24 Cyc,)
[50] iload_1 ireturn
#10 (goto)(11 By, 12 Cyc,)
[53] $1<-iload_1 iload_1 iadd
[53-52] $3<-iload_3 iconst_1 iadd
[52] goto -> 34
#11 (return)(2 By, 24 Cyc,)
[56] iload_1 ireturn
return
return
#12 EXIT

Figure 3.3: Control Flow Graph of the Java code presented in Listing 3.1

- For each vertex, the sum of f_j for the incoming edges must be equal to the sum f_k of the outgoing edges.
- The frequency of the edges connecting the loop body with the loop header is less than or equal to the frequency of the edges entering the loop multiplied by the loop bound.

There are two special vertex in the graph: The start node S and the termination node T. The start node has an outgoing edge that points to the first basic block of the method. The termination node has only incoming edges. In both types, the execution frequency of the edges is set to 1.

Loops add more constraints to the ILP problem. A special vertex, the loop header, is connected by the following edges:

(a) Incoming edges that enter the loop with frequency f_h.

(b) One outgoing edge entering the loop body with frequency f_l.

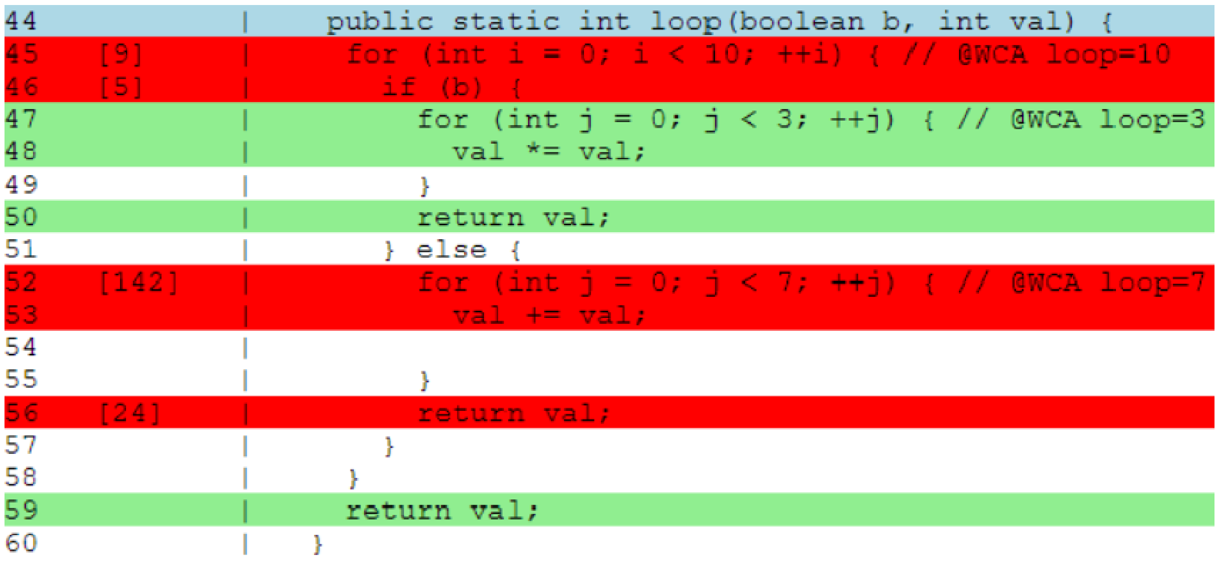
(c) Incoming edges that close the loop.

(d) One loop exit edge.

With the maximum loop count n we formulate the loop constraint as

$$f_l \leq n \sum f_h$$

Without further global constraints, the problem can be solved locally for each method. We start at the leaves of the call tree and calculate the WCET for each method.



```
44              |       public static int loop(boolean b, int val) {
45    [9]       |         for (int i = 0; i < 10; ++i) { // @WCA loop=10
46    [5]       |           if (b) {
47              |             for (int j = 0; j < 3; ++j) { // @WCA loop=3
48              |               val *= val;
49              |             }
50              |             return val;
51              |           } else {
52    [142]     |             for (int j = 0; j < 7; ++j) { // @WCA loop=7
53              |               val += val;
54              |
55              |             }
56    [24]      |             return val;
57              |           }
58              |         }
59              |         return val;
60              |       }
```

Figure 3.4: Annotated code contained in the report given by WCA. The blue line indicate the heading of a method. The red lines show the Worst Case Execution Time path, we can notice at the left side the numbers between brackets, they indicate the number of cycles the basic block takes. The green lines show the parts of the application that are not an element of the WCET.

5. Report the results. At the end of the analysis, the WCA creates a detailed report to provide feedback to the user. This report contains the callgraph, which is the graph containing the invocation of methods inside the analyzed method. Also it provides the CFG of the method and a copy of the source code of the class with provided timing annotations of the WCET (Figure 3.4).

Chapter 4

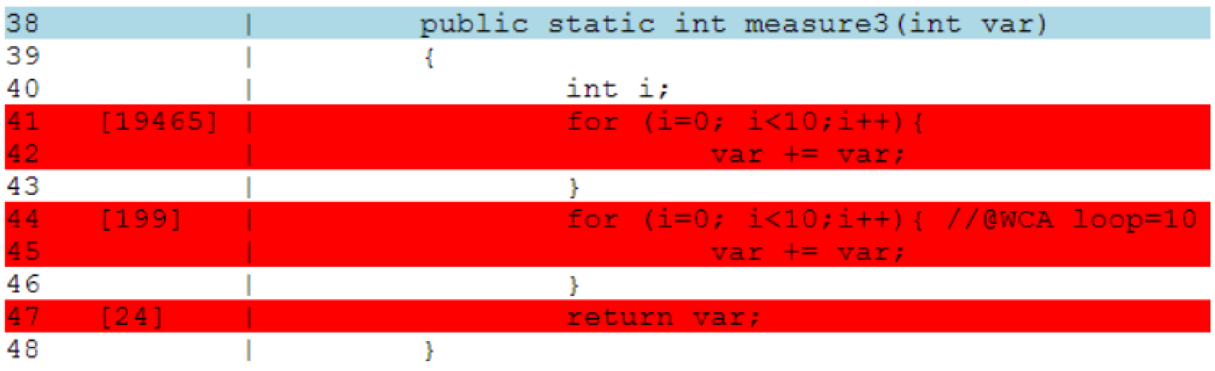
Execution time analyzer

4.1 What we learned from WCA

After having used WCA and understood its methods to obtain the WCET, we have concluded that the tool did not provide exactly what we were needing. There were properties that made us have a second thought about using WCA. However, the study of the tool also provided ideas and tools that brought a solution to our problem. The following points explain these properties:

4.1.1 Properties found executing WCA

- Default loop cycles. If the method we want to analyze is not invoked in the main method, or if there is a complex condition in the loop head of the method, then the upper bound of the loop cannot be provided by the tool. This situation happens at the moment WCA performs the data-flow analysis of the analyzed method, which provides automatic annotation of the code [17, p. 109]. As a consequence, we need to provide annotations manually to indicate the number of cycles a loop will perform. If we don't do it, a default number of 1024 cycles will be assigned to make the calculations of the WCET. Which can bring a big difference in precision of the WCET. This can be seen in figure 4.1. However, annotating the code manually brings out two issues: the first is that we are using constants, what if our condition is being made by a variable? The second one is that the user has to figure out this constant number by himself. Nevertheless, in our project we do not focus on this issues. The timed automata analysis will explore all possible number of loopings to make sure that nothing goes wrong.



```
38           |          public static int measure3(int var)
39           |          {
40           |                  int i;
41   [19465] |                  for (i=0; i<10;i++){
42           |                          var += var;
43           |                  }
44   [199]   |                  for (i=0; i<10;i++){  //@WCA loop=10
45           |                          var += var;
46           |                  }
47   [24]    |                  return var;
48           |          }
```

Figure 4.1: Result of the WCET between a non annotated loop and an annotated loop. It is evident the difference between in execution time between the first and second for loops, although they have the same statements

- WCET calculation of JOP native instructions. As mentioned in section 3.2.1, in order to transform some bytecodes into microcode, it was necessary for the JOP developers to reformulate those instructions using a set of bytecodes, meaning that these statements were implemented in Java. It was detected that operations like division and the calling for sleep method bring a higher number of cycles compared with what a single bytecode can provide. To illustrate this, the highest number of cycles a bytecode can bring is 115 from the *invokeinterface* instruction, but a division operation will bring 2169 cycles and a sleep invocation takes 149922 cycles. Then, after further checking of the report provided by WCA we could see that these two operations are implemented in Java instead of being implemented in hardware/microcode. In the case of division, the method first check if the divisor is 0 and also if the divisor is the minimum value for signed integers and only returns a constant, if it's not the case, then performs the division operation at bit level, this last part takes the WCET and that is why it returns 2169 cycles. For the sleep method it performs an operation inside a while loop containing a complex condition in its header and no manual annotation. But even if an annotation can be provided, the precise calculation of Sleep's WCET is not done, because of what we discussed in the last point.

- Sleep instruction timing ignored. Sleep is implemented with a set of instructions fitting JOP's requirements, like we mentioned in the previous point. But also we noticed by using many sleep instructions with different values that WCA is not capable to add to the WCET the number of milliseconds the sleep instruction takes as an argument. It always provide the same number of cycles, except from the first time sleep is used, even so, this first number is approximate to the rest of the results. As stated in the last point this number is very high (149922 cycles) compared to other instructions we can find in a Java program. After checking the code inside JOP we witnessed that even trying to annotate it manually, there's a loop structure inside that takes the default loop count of 1024. Thinking about this problem, maybe for the case of WCA having this kind of results is just a problem of overestimating the WCET, but for us it is unacceptable to have this kind of problem in our project.

4.1.2 Cons of using WCA

- WCA only obtains the basic blocks. In many occasions, a basic block, being a set of many bytecode instructions, can be formed by many statements of Java code. The results given in the report are then the sum of bytecode instructions in one basic block and not the sum of the bytecode corresponding to each line of Java code. In our project, several threads and the scheduler can switch from one to another in the middle of a basic block, so as we can see, this is a granularity problem. That is why we need the number of cycles each Java statement takes.

- Due to JOP's architecture, WCA assumes that methods are executed as a whole (use of the method cache) in order to improve the time predictability of the processor. However, this assumption does not match our model, again from the reason mentioned in the previous point: threads can be interrupted after any Java statement.

- The tool only provides the worst case execution time. As we could see in the annotated code of the report in 3.2.2, the code didn't provide annotations for all the basic blocks, only for those that belonged to the path having the WCET. In other words, WCA provides an analysis of the "slowest set of blocks", we don't need this, we require information that WCA process during the creation of the control flow graph but dropped for the report.

4.1.3 Pros of using WCA

- In [17, 16] and at every report of WCA we can find a table containing the reference between each bytecode and the number of microcode lines/cycles it takes to execute. In the aforementioned references we can also find how this bytecodes were implemented. We can use this bytecode-microcode reference to our advantage, by using it to calculate the number of cycles of the bytecodes in a Java code line.

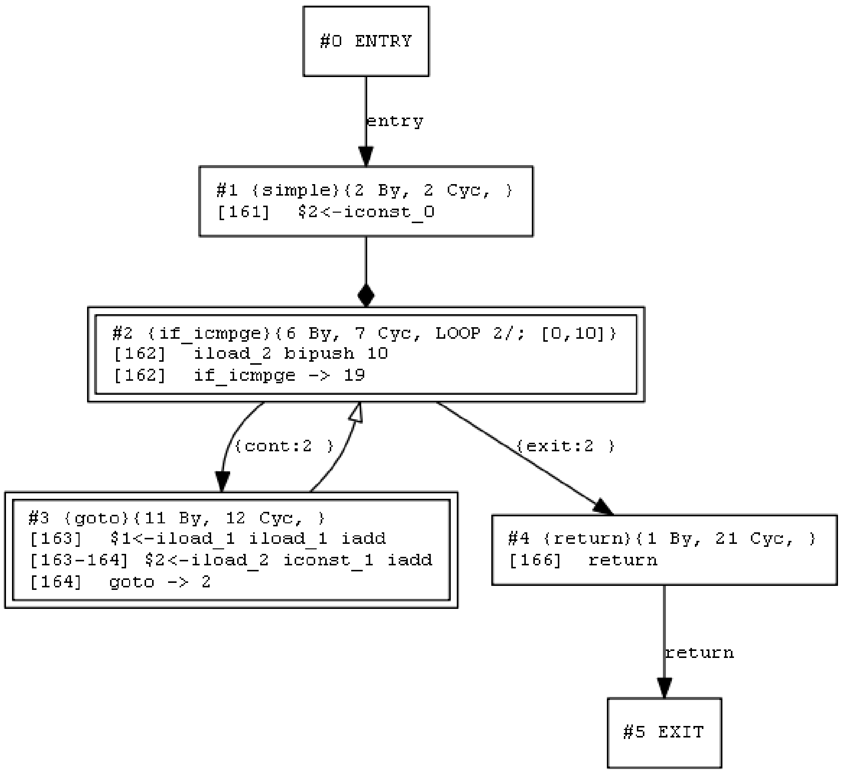

#0 ENTRY
entry
#1 {simple}{2 By, 2 Cyc, }
[161] $2<-iconst_0
#2 {if_icmpge}{6 By, 7 Cyc, LOOP 2/; [0,10]}
[162] iload_2 bipush 10
[162] if_icmpge -> 19
{cont:2 }
{exit:2 }
#3 {goto}{11 By, 12 Cyc, }
[163] $1<-iload_1 iload_1 iadd
[163-164] $2<-iload_2 iconst_1 iadd
[164] goto -> 2
#4 {return}{1 By, 21 Cyc, }
[166] return
return
#5 EXIT

Figure 4.2: Example of a CFG created by WCA, we can see the bytecode at each node, this nodes represent basic blocks

- Making an observation on the control flow graph of WCA, we can see that at each node, there are listed the corresponding lines of bytecode, we can see that in Figure 4.2. The following questions arised: how WCA was obtaining all of the bytecode and if it was possible to get the bytecode of each Java line. After making an inquiry it was determined that the byte code engineering library (BCEL) was the tool used for WCA to get all the bytecode.

4.2 Execution time analyzer

The next step in my internship was then, to provide the team with a tool that could take a Java file and give as an output another Java file with annotations of the time in cycles of each statement. The name of this prototype tool is Execution Time Analyzer (ETA).

The ETA was built completely in Java using the Java Development Kit (JDK v1.7). It has a simple user interface implemented with the swing library, it is intended to make easier the use of the tool, avoiding the command line, which was used by JOP. In order to run it properly it is necessary to have the JDK installed instead of the Java Runtime Environment (JRE) because we use a Java interface called JavaCompiler. As we may guess by its name, it provides the appropriated means to analyze syntactic correctness and to produce Java compiled files. It also can generate diagnostics during compilation, like error messages for example. Another tool that is important to remark is BCEL. As aforementioned, it is also part of WCA tools to obtain and analyze the bytecode from Java compiled files. We use it in the same way to recover and translate the bytecode from the compiled file into a more human-readable format, but then we use the data obtained in other way.

At the current stage of the prototype, the application consists of four classes. The main one, TimeAnnotationTool, is the class containing the user interface and the main functionality of the program. In Listing 4.1 we can see the code of the method containing the application flow, which consists into take the Java file and compile it. Then recover the class file using BCEL. After that, subtract every line of code from the Java file and check if these lines are analyzable, which means that they don't belong to the main method of the class. If the line is analyzable, then we pair the Java source statement with its correspondent set of bytecode and perform a sum of cycles and finally we put the annotation along with the Java code in a new Java file. This flow will be explained in detail in the next section.

```java
private void startAnalysis(){
    //Compile the code of the Java file
    boolean bSuccess = compileCode();
    if (bSuccess){
      try {
        //Recover Class file
        ClassParser oClassParser = new ClassParser(file.getAbsolutePath().replace(".java" , ".class" )) ;
        //Parse the class in order to manipulate it
        JavaClass jc = oClassParser.parse();
        Method [] aMethods = jc.getMethods();
        BufferedReader javaBufferedReader = new BufferedReader(new FileReader(file));
        BufferedWriter javaBufferedWriter = new BufferedWriter(new FileWriter(file.getAbsolutePath().replace(".java",
    "_a.java")));
        String line = null;
        int lineNumber = 1;
        //Read the Java file in order to analyze
        while((line = javaBufferedReader.readLine()) != null)
        {
          //Check if the line belongs to an analysable method
          if (isAnalysable(lineNumber, aMethods)){
            //Get the annotation by pairing the java code with the bytecode
            String sAnnotation = getTime(lineNumber, aMethods);
            //Write the java line and the associated annotation in a new file
            javaBufferedWriter.write(sAnnotation);
            javaBufferedWriter.write(System.getProperty("line.separator"));
            javaBufferedWriter.write(line);
            javaBufferedWriter.write(System.getProperty("line.separator"));
            javaBufferedWriter.flush();
          }
          else {
            javaBufferedWriter.write(line);
            javaBufferedWriter.write(System.getProperty("line.separator"));
            javaBufferedWriter.flush();
          }
          lineNumber++;
        }
        javaBufferedReader.close();
        javaBufferedWriter.close();

      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }
```

Listing 4.1: The method containing the main functionality of the system

The other three classes are InstructionTiming, JavaLine and TimingTable. TimingTable is in charge of reading the file containing the table of the relationship between the bytecode and their number of cycles. This class is also storing the information in a list of InstructionTiming objects. The Instruction Timing class is the representation of the relationship previously mentioned. The JavaLine class is the representation of each one of the Java source lines, it also contains their correspondent line numbers.

4.3 Execution stages

1. The program uses a Java source file as an input, which must be provided by the user. After this has been done we can start the analysis. The program uses the JavaCompiler tool to analyze the

code and check if there are errors. If not, the code is compiled, which gives us a .class file.

```
1  package test;
2
3  public class Test {
4
5    public static void main(String[] args) {
6      Test test = new Test();
7    }
8
9    public void foo()
10   {
11     int i = 1;
12     int j = 0;
13     for (; j < 12; j++){
14       i = i - 1;
15       i = i + j;
16     }
17   }
18 }
```

Listing 4.2: A java code example

2. Using BCEL, we recover the .class file, then we parse the hexadecimal version of bytecode to a readable one. This readable version contains a collection of methods, then it is necessary to check if the method is different from main. In the context of this work, the main method, which is static, is where we can instantiate all our classes and call their methods. As a consequence, this method is out of the scope of the program analysis. Also it is important to restrict dynamic Class loading because it doesn't guarantee a precise calculation of the WCET: we would not know what to analyze.

   As we can see in Listing 4.3, for each method we have a list of bytecode commands with a pointer at the left side. At the bottom ot this listing we have a list of attributes labeled *LineNumber* with two arguments. This attribute indicate the relation between the bytecode and the Java code, represented by their pointers at the first and second argument respectively. As an example, if we again look at the first LineNumber attribute in Listing 4.3, the first argument represents the pointer to line 0 of bytecode: *iconst_1*; the second argument represents the pointer to line 11 of Java code: *int i = 1;* (see Listing 4.2).

```
Code(max_stack = 2, max_locals = 3, code_length = 25)
0:     iconst_1
1:     istore_1
2:     iconst_0
3:     istore_2
4:     iload_2
5:     bipush    12
7:     if_icmpge   #24
10:    iload_1
11:    iconst_1
12:    isub
13:    istore_1
14:    iload_1
15:    iload_2
16:    iadd
17:    istore_1
18:    iinc      2 1
21:    goto      #4
24:    return

Attribute(s) =
LineNumber(0, 11), LineNumber(2, 12), LineNumber(4, 13), LineNumber(10, 14),
LineNumber(14, 15), LineNumber(18, 13), LineNumber(24, 17)
```

Listing 4.3: Bytecode after being parsed by BCEL

Opcode	Instruction	Implementation (over JOP)	Cycles
3	iconst_0	hw	1
4	iconst_1	hw	1
16	bipush	mc	2
27	iload_1	hw	1
28	iload_2	hw	1
60	istore_1	hw	1
61	istore_2	hw	1
96	iadd	hw	1
100	isub	hw	1
132	iinc	mc	8
162	if_icmpge	mc	4
167	goto	mc	4
177	return	mc	21

Table 4.1: Relation between the bytecode used on the example and their number of cycles

3. Having the Attributes section on the listing, we can know how many bytecode lines are being used per Java statement by taking the first argument of the next LineNumber attribute, then the first argument of the current LineNumber attribute we are analyzing. Then we make the subtraction of these two numbers. The result is the number of bytecode lines used by Java statement. As an example, if we are analyzing the first line (line 11) from Listing 4.2, then we take the number 2 from the next LineNumber attribute and subtract 0 from our current LineNumber attribute. Which gives us the first two bytecodes of the listing: iconst_1 and istore_1. These two instructions mean pushing a constant with a value 1 onto the stack and storing a value into the local variable 1 respectively. Which make sense if we look at our Java statement in line 11 in Listing 4.2: *int i = 1;*.

   Now, we know the number of bytecodes we will take for each Java line. The next step is to sum the number of cycles we will be obtaining for each bytecode until all the bytecodes from the Java statement are taken into account. We can get these values by querying the bytecode name in the TimingTable class. The TimingTable class is a representation of the previously mentioned table integrated in JOP that relates each bytecode with the number of cycles it takes to execute. The bytecode and the values being used in our example can be seen in Table 4.1. The result from the sum is the number we want to display in our output.

4. We write on the buffer the result of the sum as an annotation of the form *//@n@//* being n the result of the sum. This number will be written after the corresponding analyzed statement of Java code. After having all of the lines checked, ETA writes a copy of the original Java source file with the freshly provided annotations, as we can see on Listing 4.4.

```java
package test;

public class Test {

  public static void main(String[] args) {
    Test test = new Test();
  }

  public void foo()
  {
    int i = 1; //@2@//
    int j = 0; //@2@//
    for (; j < 12; j++){ //@7@//
      i = i - 1; //@4@//
      i = i + j; //@4@//
    } //@12@//
  } //@21@//
}
```

Listing 4.4: Annotated Java code, the annotation at line 13 belongs to the conditional statement. The number of cycles of the increment can be seen at line 16

4.4 Difficulties and challenges

During the development of the tool, my particular challenge was to become familiar with the Java programming language and working with Eclipse. Coming from C#, coding the basic control structures was not problematic, neither analyze and understand the little differences like the names of primitives and the creation of getters and setters. But I came across with some differences when handling collections of in-memory data that complicated my code writing.

Another more important difficulty we found during the development was the treatment of the for loop.

As we know, in Java the for loop header contains three statements: Initialization of a control variable, a condition that must be satisfied to enter at the loop body and an update of the control variable. We detected by testing that BCEL does not separate the first and second statements, this circumstance gives an error in the specified requirements, meaning that we cannot obtain the number of cycles of each statement. A possible solution was to find the loop header and separate the statements, placing the first one before the loop header as Listing 4.4 pictures it. In this way we can also place the annotation in separate lines, which makes clear the reading of data for the user. This operation is performed in a new Java file in order to preserve the original code. However this approach provokes an error whose origin has not been determined yet: after creating the copy of the original Java file, we try to compile it in order to get its class file, but the compiler reports and error. However, when we try to write the file manually, the compiler creates a class file successfully.

Another problem related to the for statement is to obtain the timing and also placing the annotation of the 3rd instruction of the for header due to a lack of analysis at the beginning of the development: the original approach was to check one LineNumber attribute after another and annotate in the same way the code in a new file. We assumed that the pointer to the Java code was always increasing. Nevertheless, we didn't considered the case illustrated in the sixth LineNumber attribute in Listing 4.3, the pointer of the second argument, belonging to the Java code, goes back to line 13. As a consequence, the program ignores completely the calculation of the corresponding bytecode instructions located at that line and thus, it is avoiding the annotation of the Java line.

Chapter 5

Conclusion and future work

In this document we have presented the problem of solving resource conflicts in Java systems. We introduced the subject by defining concepts around concurrency and real-time systems like an event, types of events, types of real time systems, predictability, and timing anomalies. Then we talked about the most common way to solve resource conflicts which is by using locks. However, because of well-known issues on this method, we proposed an alternative solution, consisting on using the constraints on the scheduling of parallel threads of real-time systems to ensure no conflicting access occur. To do so, we suggested the use of timing annotations at each statement of the code and use sleep statements that meet the timing obtained from the annotations and then, prove the detection of resource conflicts by translating the annotated code to a logical structure and solving the problem by using a model checker, namely Uppaal or an SMT solver.

After that, we introduced another important concept related to hard real-time systems: Worst Case Execution Time. This concept is the one we used for our code annotations and thus we established the importance to obtain a precise WCET and mentioned two methods to calculate it: measurement and static analysis. As static analysis can bring a more accurate result, we decided to study a WCET analyzer for the Java Optimized Processor called WCA.

We discussed the architecture of JOP and stated the properties that makes it a time-predictable Java processor. Some of the properties are its reduced stack instruction set, called microcode, and the way instructions are fetched into cache, by placing all the content of a method there. Then, we talked about coding restrictions in order to make WCET calculation a decidable problem and introduced our studied tool: WCA. We discussed its characteristics and the execution flow. As part of this flow analysis, we state the way WCA calculates the WCET and we mentioned observations of what we found during the study of the tool.

As aimed at the beginning of the project, based on the observations made on WCA we presented the prototype of a tool called Execution Time Analyzer that will be used by our team in the AJITPROP project. We explained each step of the system's execution and showed how to obtain an annotated Java source code by using some of the libraries found in JOP.

As a part of the future work it would be necessary to clean and separate the code of the Execution Time in order to create a better and understandable structure of the system. Also information about installation of the system will be provided. This will help the next interns working on the project by shortening the time to understand the tool's functioning.

It is also important to keep testing by using different types of structures in the Java source code and make the appropriate debugging. As part of this exercise, it is advisable to analyze the possibility to change the order we are using the components in the system, this could help solving the problem mentioned in section 4.4 where we need to change automatically the position of the first statement of the *for* loop header.

Another issue that we are interested is to develop a method that takes the number of milliseconds a sleep instruction uses as argument and combine it with the number of cycles obtained from the other type of instructions. This would help the next steps of the project by providing precise information to analyze when the translation of the code and model checking will be performed.

Bibliography

[1] Philip Axer, Rolf Ernst, Heiko Falk, Alain Girault, Daniel Grund, Nan Guan, Bengt Jonsson, Peter Marwedel, Jan Reineke, Christine Rochange, Maurice Sebastian, Reinhard Von Hanxleden, Reinhard Wilhelm, and Wang Yi. Building timing predictable embedded systems. *ACM Transactions on Embedded Computing Systems*, 13(4):82:1–82:??, February 2014.

[2] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL – a tool suite for automatic verification of real-time systems. In *Proc. Workshop on Verification and Control of Hybrid Systems*, pages 232–243, 1995.

[3] Michel Berkelaar, Kjell Eikland, and Peter Notebaert. lp_solve 5.5, open source (mixed-integer) linear programming system. Software, May 1 2004. Available at <http://lpsolve.sourceforge.net/5.5/>. Last accessed Dec, 18 2009.

[4] Nikolaj Bjørner. Taking satisfiability to the next level with Z3 - (abstract). In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *IJCAR*, volume 7364 of *Lecture Notes in Computer Science*, pages 1–8. Springer, 2012.

[5] Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe, and Pascal Fontaine. verit: an open, trustable and efficient smt-solver. In *Proc. Conference on Automated Deduction (CADE).*

[6] Markus Dahm. Byte code engineering with the BCEL API. Technical report, April 03 2001.

[7] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Kluwer Academic Publishers, Norwell, MA, USA, 1st edition, 1997.

[8] Phillip A. Laplante. *Real-Time Systems Design and Analysis.* IEEE Press and Wiley-Interscience, fourth edition edition, 2004.

[9] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 16(12):1477–1487, 1997.

[10] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *RTSS*, pages 12–21. IEEE Computer Society, 1999.

[11] Lawrence C. Paulson. The isabelle reference manual. Technical Report 283, University of Cambridge, Computer Laboratory, 1993.

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

[13] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, 2007.

[14] Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In Frank Mueller, editor, *WCET*, volume 4 of *OASICS*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[15] Martin Schoeberl. Evaluation of a Java processor. In *Tagungsband Austrochip 2005*, pages 127–134, Vienna, Austria, October 2005.

[16] Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems.* PhD thesis, Vienna University of Technology, 2005.

[17] Martin Schoeberl. *JOP Reference Handbook: Building Embedded Systems with a Java Processor.* CreateSpace, Paramount, CA, 2009.

[18] Martin Schoeberl and Rasmus Pedersen. WCET analysis for a Java processor. In *Proceedings of the Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2006)*, pages 202–211, Paris, France, October 2006. ACM.

[19] Martin Schoeberl, Wolfgang Puffitsch, Rasmus Ulslev Pedersen, and Benedikt Huber. Worst-case execution time analysis for a java processor. *Softw, Pract. Exper*, 40(6):507–542, 2010.

[20] Inc. Sun Microsystems. Jsr 302: Safety critical java technology. http://jcp.org/en/jsr/detail?id=302 (Date retrieved: March 19, 2008), 2007.

[21] Lothar Thiele and Reinhard Wilhelm. Design for time-predictability. In Lothar Thiele and Reinhard Wilhelm, editors, *Design of Systems with Predictable Behaviour*, volume 03471 of *Dagstuhl Seminar Proceedings.* IBFI, Schloss Dagstuhl, Germany, 2004.

[22] Sascha Uhrig and Jörg Wiese. jamuth: an IP processor core for embedded java real-time systems. In Gregory Bollella, editor, *JTRES*, ACM International Conference Proceeding Series, pages 230–237. ACM, 2007.

[23] Reinhard von Hanxleden, Niklas Holsti, Björn Lisper, Erhard Plödereder, and Reinhard et.al. Wilhelm. WCET tool challenge 2011: Report. In *Proceedings of the 11th International Workshop on Worst-Case Execution Time (WCET) Analysis; Porto, Portugal*, Artikel in Tagungsband, pages 1–38. -, July 2011.

[24] Reinhard Wilhelm, Sebastian Altmeyer, Claire Burguière, Daniel Grund, Jörg Herter, Jan Reineke, Björn Wachter, and Stephan Wilhelm. Static timing analysis for hard real-time systems. In Gilles Barthe and Manuel V. Hermenegildo, editors, *VMCAI*, volume 5944 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2010.

[25] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):36:1–36:??, April 2008.