Execution Time Analyser

Yashas Nagaraj Udupa

2019-02-13

Abstract

A real-time system must react within precise time constraints, related to events in its environment and the system it controls. This means that the correct behavior of a real-time system depends not only on the result of the computation but also on the time at which the result is produced. Thus, knowing the execution-time characteristics of a program is fundamental to the successful design and execution of a real-time system. Reliable timing estimates are important when designing and verifying many types of embedded systems and real-time systems. This is especially true, when the system is used to control safe critical products such as vehicles, aircraft, military equipment and industrial plants. Depending on the criticality of the system, a coherent analysis might be acceptable. The determination of upper bounds on execution times, commonly called Worst-Case Execution Times (WCET), is a necessary step in the development and validation process for hard-real time systems. The worst-case execution time of a program or subprogram can be found in two ways namely, Static WCET analysis, in which the application is analyzed off-line to produce the maximum running time. And Measurement based WCET analysis, in which the application run on the target system or a processor simulator and the WCET is measured. Unfortunately, measurements do not give any guarantees that the longest execution time has been found and is often time consuming and error-prone. Static WCET analysis appears to be a reasonable one since methods work by analyzing the program, instead of running it, to derive properties that will hold for all possible program runs. Open Toolbox for Adaptive WCET Analysis(OTAWA) plays a good role in obtaining WCET because of the fact that it is able to perform and combine several different analyses together to derive overall WCET of a task.

Contents

List of Figures

1 Introduction

This document describes the WCET analysis framework for OTAWA to the target processors. OTAWA is not a tool but a toolbox, which mainly comes as a C++ library that can be used to develop WCET analysis tools. However, it includes a number of algorithms which make this task really easy. The general approach to WCET analysis includes three steps: (a) flow analysis mainly consists in identifying (in) feasible paths and bounding loops; (b) low-level analysis aims at determining the global effects of the target architecture on execution times and at deriving the worst-case execution times of code snippets; (c) finally, the results of the flow and low-level analyses are combined to derive the overall WCET.

2 Implementation Architecture

OTAWA exhibits two key features. First, both the target hardware and the code under study are processed through an abstract layer: this way, analyses are designed independently from the hardware and from the instruction set which can then be used for architecture. Second, the toolbox provides all the facilities to implement new analyses with limited effort.

2.1 Abstract layers

OTAWA is organized in independent layers that provide an abstraction of the target hardware and associated Instruction Set Architecture (ISA) as well as representation of the binary (.elf) code under analysis(irrespective of the elf format, the tool considers for the analyses i.e ELF32 or ELF64).

2.2 ISA support

This layer hides the details of the target ISA to the rest of the toolset and provides a unique interface to retrieve the information needed for the analyses related to WCET estimation. This way, most of the components of the toolbox can be designed independently of the instruction set. The layer features support for various ISAs implemented through a set of plugins. Available plugins have been generated using GLISS2 tool.

2.3 Abstraction of the hardware architecture

Precise estimation of the worst-case estimation of instruction sequences requires a detailed knowledge of hardware parameters:
e.g: processor - width and length of the pipeline, number of functional unnits and their latencies, binding of instruction categories to the functional units, specification of the instruction queues (location, capacity).

3 Installation

This section describes the requirements to build and use the OTAWA[6] framework. To build OTAWA, the following packages have to be installed:

3.1 For Ubuntu Users :

The below table provides information about the tool and the corresponding commands to install associated packages.

Tool	Package
GCC	`sudo apt-get update`
	`sudo apt-get install gcc g++`
gnumake	`sudo apt-get install build-essential`
cmake	`sudo apt-get install cmake`
flex	`sudo apt-get install flex`
Bison	`sudo apt-get install bison`
libxml2	`sudo apt-get install libxml2`
	`sudo apt-get install libxml2-dev`
libxslt	`sudo apt-get install libxslt1-dev python-dev`
git	`sudo apt-get install git`
Mercurial	`sudo apt-get install mercurial`
Java	`sudo apt-get install default-jdk`
OCAML	`sudo apt-get install ocaml`

Further dependencies can be met inside the machine by doing the below procedure.

Procedure:

- Download the following package **otawa-build.tgz**[3],
- unpack it,
- move to the unpacked directory,
- launch command: `./build.sh`,
- Set the environment variables $PATH to OTAWA features inside the bash script i.e.
  - Open in command line terminal and enter `gedit ~/.bashrc`.
  - Export the path by writing
    export PATH=/home/otawa/OTAWA_FOLDER/otawa/linux-x86-64/otawa-core/bin/:$PATH

There are also extensions that are to be built once the OTAWA packages are installed. To extend OTAWA facilities, some analyzes are packaged in plugins. Some plugins are devoted to load and decode binaries, namely:

- ARM v7(experimental),
- PowerPC,
- Sparc,
- TriCore,
- Star12X.

For the exclusive benefit of testing and validating the tool OTAWA, the plug-in ARM v7 is being considered. And below are the commands that are to be in executed in the terminal in the mentioned order to install ARM v7 plugin:

Commands[1]	`hg clone https://anon:anon@www.irit.fr/hg/TRACES/lpc2138/trunk lpc2138`
	`cmake . -DINSTALL_TYPE=all -DCMAKE_INSTALL_PREFIX:PATH=/home/yashas/otawa/linux-x86_64/otawa-core`
	`make install PREFIX='/home/yashas/otawa/linux-x86_64/otawa-core' INSTALL_TYPE=all user=None`

1. The files which were imported faced an issue with typos in the code which, manually had to be resolved. But the latest version of the packages are being fixed and updated in the repository by developers.

Although OTAWA is designed as a generic framework to develop static analyses for WCET computation, some facilities are provided to compute WCET using dedicated commands. OTAWA implements several approaches to compute Worst Case Execution Time, in where it mainly makes use of a principle called as IPET(Implicit Path Enumeration Technique)[5]. There are four ways to compute a WCET, they are:

- oipet command provides minimal support for IPET computation and may be used for basic computations,
- owcet command allows to perform computation based on script adapted to specific architecture,
- The Eclipse plug-in gives access to architecture scripts in more friendly graphic way inside the Eclipse IDE,
- By writing your own program benefiting from the facilities of the framework.

Computation of Worst Case Execution Time can be effeciently done by equally benifiting the combined features of OTAWA plug-in in Eclipse IDE and command line environment of OTAWA. OTAWA plug-in in Eclipse enables to :

- to run OTAWA on the developed binary,
- to display the internals of the library,
- to select the tasks of the project,
- to help bounding the loops of the project (by hand or using oRange),
- to compute the WCET,
- to visualize time spend on sources, on assembly, on CFGs or the PCG.

Steps that are involved to obtain Eclipse[2] version of OTAWA:

1. Download **Otawa-plugin** for Eclipse.
2. Extract the plug-in at the desired folder.
3. Install latest/galileo version of Eclipse by following the instructions present in this **link**,
4. Copy the 'dropins' folder of the plug-in to ECLIPSE_HOME by entering following commands,
   - cd otawa_eclipse_plugin.
   - cp -R dropins ECLIPSE_HOME.
   - Set the toolchain path environment in bash script by writing : export PATH=/home/otawa/ARM_TOOLCHAIN_PATH/ bin/:$PATH
   - Finally, open Eclipse and compile using ARM Tool chain. Subsequently, the generated .elf file can be analysed via Eclipse or command line terminal using OTAWA utilities.

2. Eclipse plug-in is not fully functional because of the haze structure of plugin causing problems to operate oRange[2] and also inconvenience in invoking/installing architecture specific OTAWA script within the plug-in.

3. Words in bold letters denote that it can direct to the supporting URL.

3.2 Functional Test :

To test, ensure and get the sense of utilities and end to end functionality of OTAWA[4] tool, simple fibonacci program's executable(.elf) file is being considered, which is being built using ARM tool chain. Below are the steps which need to be carried out sequentially to get WCET result:

1. Enter the command `oipet executable_path fib`, to approximmately get the execution time, which does not make use of architecture specific information but just "Implicit Path Enumeration Technique".
2. Enter the command `owcet -s SCRIPT EXE_FILE [TASK_NAME] -f PATH_TO_FLOWFACT_FILE` to obtain WCET based on specific architecture information. But before executing this command flowfacts, name of the script and the desired tasks' information need to be obtained and provided.
   - By entering the command `otawa-config --list-scripts`, list of available scripts can be obtained and thereby, keyword `SCRIPT` can be replaced with the desired one. e.g. lpc2138.
   - `TASK_NAME` for an instance, can be replaced by 'fib'.
   - `EXE_FILE` can be replaced by an executable path of fibonacci program.
   - `PATH_TO_FLOWFACT_FILE` to be replaced with flow-fact path which firstly, expects for flow facts to be produced.
     - Flow facts can be produced by entering the command `mkff executable > executable.ff`.

4. It can be tested either by virtual machine image of OTAWA which is uploaded in the shared repository **MP_Udupa_WCET .git** where the code is present in eclipse-workspace folder (home/otawa/ eclipse-workspace/) and OTAWA tool is present in Downloads folder (home/otawa/ Downloads/)

  - And in the generated flowfact file (executable.ff) to provide information about loop bounds (Either by running oRange or by manually inserting).
    e.g. loop 0x0005013c 30;

- After replacing the keywords with actual information, the command is executed to obtain WCET result.
  e.g `owcet -s lpc2138 /../fibonacci/Debug/executable.elf fib -f /../fibonacci/Debug/executable.ff`

4 Features

OTAWA is actually a collection of several related tools. Both the command line tools and the Eclipse plugin can be downloaded as binaries for Windows and Linux. To compute WCET in static and sound approach, one has to consider the running hardware and the executed program. OTAWA contributes by providing: (a) a set of tools to compute WCET for some hardware platforms and (b) a framework to develop new analyses for new hardware or for experimental purposes. OTAWA provides follwing commands to perform analyses on the desired executable (.elf) file.

- dumpcfg command - to output Control Flow Graph at a required format (e.g .dot).
- odfa command - to examine data flow analysis on C code.
- odisasm command -disassmebling of the program in machine code as to get a view of the program as understood by OTAWA.
- oipet command - command-line tool for WCET calculation which takes a binary file and one or multiple entry points. So, one can provide a *Flow Facts* file where the loop bounds of the binary are specified.
- owcet command - to compute WCET from a script describing an hardware.
- orange Command - oRange is a command-line tool for determing loop bounds that work on source code. Unfortunately, it produces XML ouput, which is incompatible with oipet. There is no tool for covering oRange's output into a *Flow Facts* file as required by oipet, but it can be done by hand. In the command line terminal enter orange --help to know it's syntax.
- ostat command - computes some statistics about instructions of a program.
- mkff command - to generate flow fact file ready to be filled by the user from binary code.

- otawa-config --help command - gets building information about the OTAWA framework.

OTAWA is logically split into different modules representing different layers and usage of the framework. In the following, you will find a guide to explore these modules(i.e libraries comprising of OTAWA member functions and variables), which are being made use hierarchically to perform the analyses:

- **Program Representation**
- **Properties System**
- **Processor Layer**
- **Platform Description**
- **Data Flow Analysis**
- **IPET Method**
- **Flow Facts**
- **Graph**
- **ODisplay Library**
- **Application Writing**
- **Abstract Syntax Trees**
- **Branch Prediction**
- **Instruction Cache Analysis**
- **CFG**
- **Data cache**
- **Event Time**
- **Extended Timing Scheme**

5 Concept

Worst-Case execution time is typically computed running through static analysis but, to obtain precise estimation of worst-case (best-case) execution time of any given application, the detailed knowledge of hardware parameters are required, such as:

- processor: width and length of pipeline, number of functional units and their latencies, binding of instruction categories such as `MUL,DIV, CONTROL` etc to the functional units, specification and creation of the instruction queues (location, capacity), creation of stages.

- memory: write access, loading from/to registers, address allocation etc.

The factors that affect the execution time depends on the hardware such as the characterisitic of the processor, the memory hierarchy, the infrastructure of the bus, and the pheripherals (i.e inputs and outputs). In order to obtain the information about the hardware influence to the application evidently, the entire stages of operations inside the microprocessor (MSP430) need to be simulated and coherently must have to be fed to the OTAWA to collaborate with its static analyses. And it involves four levels of implementation at the mentioned order namely,

1. **Loader defintion** - OTAWA loader to support MSP430x Family's ISA with GLISS2.
2. **Instruction set simulators** - description of instruction set architecture(ISA) of MSP430 (RISC) using nML[3].
3. **Structural simulators** - simulations of functional units of CPU such as fetch, decode and execute.
4. **OTAWA Script** - allows to configure WCET computation via XML file description.

5.1 Flow of the simulation

Figure 1 illustrates step-wise operations of simulation. And it is described as follows:

1. Loader needs to be implemented, which is mainly providing architecture abstraction to OTAWA for particular ISA.
2. The structural simulator (SS) sends a request to obtain the instruction to fetch from the instruction set simulator (ISS)[7] .
3. For obtaining the first instruction of the program, the ISS will provide it directly from the execution binary. For the later instructions, the ISS will execute the current instruction and determines the next instruction to provide.
4. The SS now places the received instruction to Fetch stage.
5. In the next cycle, the instruction in fetch stage will be transferred to the decode stage. Meanwhile since the stage is empty, it will request a new instruction as mentioned from step 2 to step 4.
6. Similar to step 5, the instruction in the decode stage will be placed into the execution stage. In the execution stage, the number of the cycles required to finish the instruction will be determined according to the opcode and operands of the instructions. This will inturn stop the SS to request new instruction from the ISS. Once the instruction completes

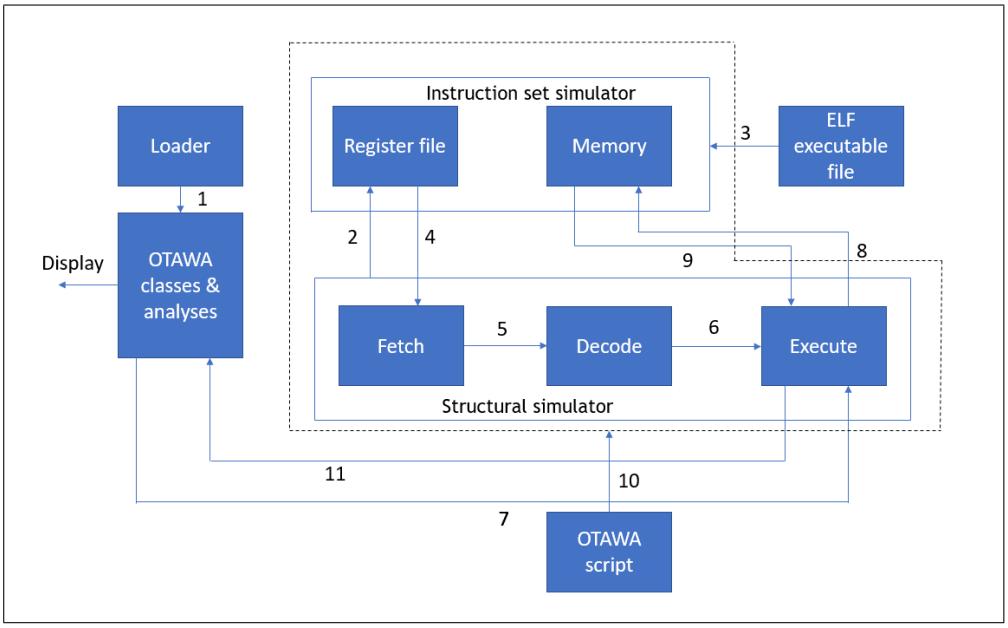

Figure 1: Simulation flow.

in cycles, the execute stage will be emptied, and the instruction will be transferred from the decode stage, as well as from the fetch stage to the decode stage. In this case, new instruction will be requested from ISS as described in step 2.

7. When the simulated program requires input from the pre-developed features(e.g owcet,mkff) of OTAWA, it is being read from the directory of OTAWA fuctionalities.
8. Through the interface of ISS, SS transfers the input data to the specified memory address.
9. Similary when the ouptut is generated from the simulation program, SS reads the data from memory.
10. OTAWA script to configure the WCET computation for a particular architecture via XML format.
11. Output is directed to OTAWA, to display the result.

6 Implementations and it's details

Following are the essential modules that are to be implemented in order to provide compatability to OTAWA in computing worst-case (best-case) execution time of MSP430x Family applications, with their details:

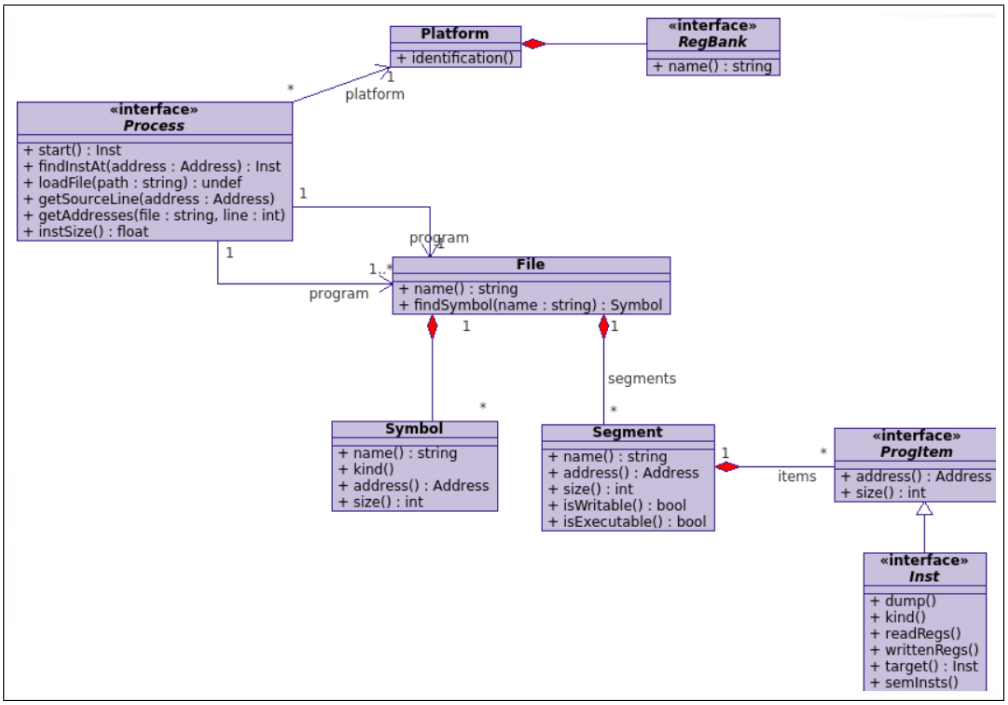

Figure 2: UML diagram[1] of the relationship between classes to implement architecture abstraction.

6.1 Loader

Developing a loader is mainly providing actual implementation architecture abstraction for a partiular ISA. In otawa loader is basically a plugin whose handle implements `otawa::Loader`. The handle object is used to load a binary file and to build the process representing the program. Implementation of a loader is done using pre-defined OTAWA C++ classes as shown in Figure 2 where, the root class is the `Process:` that represents the program ready to run. In addition, the `Process` contains also information about the programming model of the architecture i.e by an object named as `Platform`: which consists of mainly the list of register banks with their description. A class by default `Symbol:`, contains bits composing the program image in memory but also more functional information like `Symbol:`. The `Symbol:` matches any object with a name produced by the compiler, that is, function, data, labels, etc. The symbols may occupy a place in the memory (defined by its address and its sizes) or not. The `Segment:` objects represents slices of memory sharing common properties. Closer concepts from the ELF file format are either sections or program headers. A `Segment:` usually has a name, an address, a size and may be executable. They are composed of `ProgItem` object. A `ProgItem` object represents any atomic entity in the program, mainly, instructions and data items. The current version only supports instructions,

`Inst` class. The `Inst` objects represernts the actual machine instructions of the given ISA. The role of these objects is to give an abstract description of machine instructions as precise as possible to let the higher level analyses to work with the instruction. An `Inst` object have the following member functions:

- dump - to get a textual representaion,
- kind - information about the nature of instruction,
- readRegs - set of read registers,
- writtenRegs - set of written registers,
- target - when the instruction is a control, the target of the branch,
- semInsts - translate the instruction into semantics instructions.
- IS_COND - set if the instruction is conditional,
- IS_CONTROL - set if the instruction performs a branch, that is, it changes the program counter ,
- IS_CALL - set if the instruction is a sub-program call (manipulated when IS_CONTROL is set),
- IS_RETURN - set if the instruction is a sub-program return (manipulated when IS_CONTROL is set),
- IS_TRAP - set if the instruction performs a system trap like exception or system call (manipulated when IS_CONTROL is set),
- IS_MEM - set if the instruction performs memory access.
- IS_LOAD - translate the instruction into semantics instructions.
- And much more functions, which can be viewed using this **link**

The more interesting part is in create() method that build a platform of types mips::Platform and the process itself of type mips::Process. When a binary is opened from an ELF file, OTAWA looks for the matching plugin using ISA field of ELF header, install the loader and call method load(). This one creates a process and load the program in the process.

6.2 Instruction Set Simulator

The Instruction set simulator (ISS) and disassemblers are automatically generated by a tool named GLISS2[5], given the information of the MSP430x Family's instruction set architecture(ISA). The information is represented by describing the entire ISA of MSP430x with very commonly used architecture description language called SimNML. SimNML[4] describes the ISA as a collection of types, state items, modes and operations. The ISS maintains the state of the processor in a register file (set of all registers in the processor) and the content of the memory accessed by the processor. The ISS is implemented at the functional level instead of the cycle-accurate level. An nML description

5. For more details to check the documentation of GLISS2 at the following path: ../otawa/linux-x86_64/otawa-core-build/gliss2/doc/gliss-tuto.thot

is made of three parts. The first part describes the state of the system (memory and registers). The second part describes the addressing modes in order to give explicit description everytime whenever the instructions are used. The third part is the description of the instructions themselves. Considering the description of instruction set architecture of MSP430x, the register banks are being set and initialized as shown in the Figure 3. For a concrete understanding, if one register among the register bank is considered to get a sense of the description, `reg r4[1, u16] alias = R[4]`, conveys that the register is of "unsigned sixteen" bit integer and is being duplicated by a keyword 'alias' to other name `R[4]`. Similarly, `reg CFLAG[1, u1] alias = SR<0..0>` where the first bit of SR is being initalized with the value of 1 bit register `CFLAG`. And then comes the description of addressing modes where the keyword `mode` matches the usual address mode found in most processors as shown in Figure 4. In NMP, it is a way to factorize state access behaviour. All attributes are derived from these parameters which includes:

- `syntax` : assembly form of the instruction,
- `image` : binary of the instruction,
- `action` : semantics of the instructions.

To describe every data processing instructions that are involved in ISA of MSP430x primarily, a key word by name `op` is used. And every such instruction is being operated by representing them with `syntax`,`image` and `action`. The attribute `action` acts as a defintion of the operation which can be written using macro. Figure 5 shows a simple example of the representation of an instruction. Where, `image` attribute is being exclusively denoted bit-wise indicating first four signficant bits as `opcode` of the instruction, subsequent bits as image of source and destination register and bits from 8 to 4 are reserved for special purpose.

```
// MSP430 state implementation for Fast ISS

// activate only marked registers for debugging
let gliss_debug_only = 1

// register file
reg R[16, u16]

reg r4[1, u16] alias = R[4]
reg r5[1, u16] alias = R[5]
reg r6[1, u16] alias = R[6]
reg r7[1, u16] alias = R[7]
reg r8[1, u16] alias = R[8]
reg r9[1, u16] alias = R[9]
reg r10[1, u16] alias = R[10]
reg r11[1, u16] alias = R[11]
reg r12[1, u16] alias = R[12]
reg r13[1, u16] alias = R[13]
reg r14[1, u16] alias = R[14]
reg r15[1, u16] alias = R[15]

reg CG2[1, u16] alias = R[3]

reg SR[1, u16] alias = R[2]
reg CG1[1, u16] alias = R[2]

reg SP[1, u16] alias = R[1]

reg PC[1, u16] alias = R[0] pc = 1

reg NPC[1, u16]

// access to SR flags
reg CFLAG[1, u1] alias = SR<0..0>   //carry flag
reg ZFLAG[1, u1] alias = SR<1..1>   // zero flag
reg NFLAG[1, u1] alias = SR<2..2>   // Negative flag
reg GIE[1, u1] alias = SR<3..3>     //General Interrupt Enable
reg CPUOFF[1, u1] alias = SR<4..4>   //CPU Off
reg OSCOFF[1, u1] alias = SR<5..5>  //Oscillator Off
reg SCG0[1, u1] alias = SR<6..6>  //System Clock Generator 0
reg SCG1[1, u1] alias = SR<7..7>  //System Clock Generator 1
reg V[1, u1] alias = SR<8..8>  //Overflow bit
reg RSCG1[1, u7] alias = SR<15..9>  //Reserved for CG1
```

Figure 3: Description of register banks.

```
mode DIRECT(i: REG_INDEX) = R[i]
    image = format("%4b", R[i])


mode INDEXED(r: DIRECT(i),d: u16) = M[r + d]
    syntax = format("%d(r%d)",d,i)
    image = format("%4b", M[r + d])

mode SYMBOLIC(r: DIRECT(i)) = M[r + PC]
    syntax = format("%d", i)
    image = format("%4b", i)

mode ABSOLUTE(a: u20) = M[a]
    syntax = format("&%d", a)
    image = format("%16b", a)

mode INDIRECT(r: DIRECT(i)) = M[i]
    syntax = format("@%d", i)
    image = format("%4b", M[i])

mode INDIRECT_AUTO(r: DIRECT(i)) = M[i] + 1
    syntax = format("@%d+", i)
    image = format("%4b", M[i] + 1)

mode IMM(n: u16) = n
    syntax = format("#%d", n)
    image  = format("%16b", n)
```

Figure 4: Description of address modes.

Finally, they are organized in a so-called AND-OR tree as shown in Figure6. This OR operation gives alternatives for building an instruction. Once the NML file has been written, it is saved in a file named "architecture_name.nmp". And then simulator and disassembly sources helps "gep" module to generate simulator and disassembler.

```
macro ADDC(dest,src) = \
        TMP_REG1 = Get_MSP_GPR(dest);\
        TMP_REG2 = Get_MSP_GPR(src);\
        TMP_REG3 = TMP_REG2 + carry;
        TMP_REG1 = TMP_REG2 + TMP_REG1 + carry; \
        Set_MSP_GPR(dest,TMP_REG1);\
       if SBIT(TMP_REG1) == 1 then \
           NFLAG = TMP_REG1<15..15>; \
       else \
           NFLAG = 0;\
       endif;
       if TMP_REG1 == 0 then \
           ZFLAG = 1; \
       else \
               ZFLAG = 0; \
           endif; \
       CFLAG = CarryFromAdd(TMP_REG3,TMP_REG1,TMP_REG1);\
       VFLAG = OverflowFromAdd(TMP_REG3,TMP_REG1,TMP_REG1);


op ADDC(Rsrc : REG_INDEX, src_cond : card(3),
        Rdst : REG_INDEX, dst_cond : card(3), CG_choice : CGCHOICE, AS : card(2))
    syntax = format("ADDC %s,%s", Rsrc.syntax, Rdst.syntax)
    image = format("%0110%4b0000%4b",DIRECT(Rsrc).image,DIRECT(Rdst).image)
    action = {
        ADDC(DIRECT(Rdst),DIRECT(Rsrc));
    }
```

Figure 5: Description of instructions.

```
let proc = "msp"

// ****** includes ******
include "simpleType.nmp"
include "state.nmp"
include "tempVar.nmp"
include "modes.nmp"

include "condition.nmp"
include "dataProcessingMacro.nmp"
include "dataProcessing.nmp"

// ****** instruction set ******


op MSP_instr = dataProcessing


op dataProcessing = MOVA_1 | MOVA_2 | MOVA_3
                  | MOVA_4 | MOVA_5 | MOVA_6
                  | MOVA_8 | CMPA_1 | ADDA_1
                  | SUBA_1 | MOVA_7 | CMPA_2
                  | ADDA_2 | SUBA_2 | ADDC
                  | SUBC | DADD | BIT
                  | BIC | BIS | XOR | AND
```

Figure 6: Description of organisation of instructions.

6.3 Structural simulators

Structural simulators (SS) is in charge of obtaining the cycle counts of the program execution. The structural part can be described in a language more adapted like system C and can use functions provided by the ISS generator. The SS[8] is chosen to be modeled in SystemC[6], this is because the notion of the clock cycles so that the behaviours of instruction moving from one processor stage to another at each cycle can be easily captured. Using OTAWA classes the CPU's fundamental pipelining stages, syscalls, fetch, decode, execute etc need to be implemented and also must have to be synchronized with ISS to simulate the processor operations in a structured manner.

6. For more details about the organisation and implementation of SS, to check the pre-built SS of ARM processor which can be found in the below path ../otawa/linux-x86_64/ otawa-core-build/armv7t/src .

6.4 OTAWA script

The SS is configured via XML files which describe the charactersitics of the processor, such as the pipeline stages and the organization of the memories (i.e the memory types with its corresponding addresses). From the user side, the hardware parameters can be specified with an XML file. This script itself provides way to do several analyses to get the worst-case (best-case) execution time. And the code processor's class definition and the definition of its members, a small text describes the function and the algorithm of the code processor. And, the written script directly works on the code processor structure of OTAWA that is composed:

- code processor performs the analysis using available annotations.
- features are required or provided by the code processor and represent information retreived by the analyses.
- list of configuration properties representing results of analyses, hooked to the program representation and grouped in features.

It is evident that the order of feature requirements or code processor invocations is definitely a requirement. In determination of worst-case (best-case) execution time of a task involves a number of analyses that must be performed in an appropriate order, In OTAWA, such analyses are implemented as Code processors[7]: a code processor is a function that uses available annotations and produces new annotations. The distributed OTAWA library includes a set of analyses. Some are related to flow analysis and produce information that are useful to other analyses. Examples in this category are a CFG builder, a CFG virtualizer that produces a virtual CFG with the functions inlined (this allows call-contextual analyses) or a loop analyzer that determines loop headers and dominance relationships which are used e.g. by the instruction cache analyis. Other code processors provide facilities to use state-of-the art analysis techniques, like Abstract Interpretation. Finally, a number of analyses compute data that can be combined to determine the final worst-case(best-case) execution time.

7. For more details about the organisation and implementation of code processors, to check the pre-built OTAWA script of ARM processor which can be found in the below path ../otawa/linux-x86_64/ otawa-core-build/lpc2138/ .

Another use of the script is to provide support in OTAWA for a new micro-architecture which allows to describe the components of an architecture (pipeline, memory, cache). To know about the script format and their organization, they can be viewed using this **link**.

7 Progress

Here are the standings in terms of providing OTAWA compatibility support to MSP430x Family applications.

Content	**Modules**	**Remarks**
OTAWA	Functionality	Successful
	Features	Tested and Validated
Instruction set simulator	Architecture description using nml	Completed
	Simulators	Yet to implement
	Disassemblers	Yet to implement
Structural simulator and Loader	Fetch, Decode and Execute	Yet to implement
OTAWA script	XML script and code processors	Yet to implement
Best case analysis		Yet to implement

References

[1] http://www.tracesgroup.net/otawa//doc/manuals/dev/dev-1.html.

[2] Armelle Bonenfant Marianne de Michel and Pascal Sainrat. orange: A tool for static loop bound analysis. September 1, 2008.

[3] Markus Freericks. The nml machine description formalism.

[4] R. Vaillant H.Casse J. Barre and P. Sainrat. Fast instruction-accurate simulation with simnml. *Universite de Toulouse, Institut de Reche en Informatique de Toulouse(IRIT).*

[5] Isabelle PUAUT. Worst-case execution time (wcet) estimation: from monocore architectures to multi-core architectures.

[6] Clement Ballabriga Hugues Casse Christine Rochange Pascal Sainrat. Otawa: an open toolbox for adaptive wcet analysis. *Institut de Reche en Informatique de Toulouse.*

[7] Tahiry Ratsiambahotra Hugues Casse Pascal Sainrat. A versatile generator of instruction set simulators and disassemblers. *IRIT - Universite Paul Sabatier de Toulouse, Hipeac European Network of Excellence.*

[8] Wei-Tsun Sun. A framework for simulate syncronous reactive programs and measure times to aid wcet analysis. *Verimag Research Report n^o TR-2016-3*, 2016-07-20.