Exercise Sheet 1

Using OTAWA framework

H. Cassé <casse@irit.fr>

OTAWA is an open-source framework delivered under LGPL license. It is dedicated to binary code analysis and particularly to WCET computation.

Before starting Resources of **OTAWA** are located in `/nfs/otawa`. This consist in:

1. `/nfs/otawa/site/bin` – **OTAWA** commands (add this directory to your `PATH`).
2. `/nfs/otawa/labwork1.tgz` – files for this labwork (unpack it in your home directory – also available on Moodle).

Unpacking the archive `labwork1.tgz` creates a main directory named `labwork1` that contains a subdirectory for each exercise. The corresponding directory is provided at the head of each exercice. In addition, the main C source file is also provided.

The exercices of this labwork have to performed in a command line terminal!

Assignment A text report about the labwork has to be submitted to a Moodle repository. This repository contains also a template for the report. In this document, each data need to be reported is suffixed by ✎.

1 Starting up with OTAWA

Directory: `labwork1/bs`
Source: `bs.c`

The "bs" benchmark is a small application made of two functions, **main** and **binary_search**, that performs a binary search of an integer in an array. It contains only one loop in **binary_search** that has to be bounded if we want to obtain a WCET.

This small application allows us to introduce the process from the source to the WCET calculation passing by the compilation for bare metal architecture and the determination of loop bounds.

1.1 Calculating the WCET

1. Move to directory `labwork1/bs`.
2. Compile the application with the command `make`.

   Notice the options used to perform the compilation:

   -g3 Enable debugging information (useful to keep link of the binary with sources).

   -static Enable stand-alone compilation (the compiled binary is independent of any shared library).

   -nostartfiles The actual startup of the program is included in itself: no need for Linux startup.

   The produced binary file is named `bs.elf`.
3. Compute the WCET for the `main` function using the architecture `lpc2138`

   ```
   owcet -s lpc2138 bs.elf
   ```

   `lpc2138` is a family of small ARM-based microcontroler delivered by NXP. Its documentation can be found in `/nfs/otawa/lpc2138.pdf`. The family is composed of `LPC2131`, `LPC2132`, `LPC2134`, `LPC2136` and `LPC2139`, the last one being the more powerful.

   At this point, the `owcet` should have failed with a message like:

   ```
   INFO: plugged otawa/lpc2138 (.../lib/otawa/otawa/lpc2138.so)
   WARNING: otawa::util::FlowFactLoader 1.4.0:no flow fact file
           for bs.elf
   WARNING: otawa::ipet::FlowFactLoader 2.0.0:no limit for the loop
           at binary_search + 0xb4 (000101b4).
   WARNING: otawa::ipet::FlowFactLoader 2.0.0: in the context
           [FUN(000101dc), CALL(000101e8), FUN(00010100)]
   ERROR: otawa::Weighter (1.0.0):cannot compute weight for loop
           at BB 2 (000101b4)
   ```

   This means that a loop bound is missing (at address $0x101b4$).

4. To provide the missing loop bound, you have to generate a flow fact file with the command `mkff`.

   ```
   mkff bs.elf > bs.ff
   ```

   Then, you have to edit the file and replace the '?' mark with syntax "`max` N", with N being the loop bound, actually 10. The fixed `bs.ff` is displayed below:

   ```
   checksum "bs.elf" 0xc39d57a1;

   // Function binary_search (bs.c:82)
   loop "binary_search" + 0xb4 10 ; // 0x101b4 (bs.c:88)
   ```

   Notice that the flow fact file contains all required detail to find back the corresponding loop in the source: the container function, `binary_search`, and the source file and line, `bs.c:88`.

5. Now, we can relaunch the WCET computation that succeeds (the number of cycles may be different than below as this depends on the used compiler version).

   ```
   owcet -s lpc2138 bs.elf
   WCET[main] = 444 cycles
   ```

1.2 Details about the WCET

Getting the WCET is required to check the schedulability of a real-time system but what to do if the WCET does not allow to schedule the system? Maybe, we have either to optimize the tasks, or we have to change the system.

For the latter case, **OTAWA** cannot help but the former case can be guided by a more detailed understanding on where the time is spent. To get this details, we have to ask **OTAWA** to produce statistics about the WCET and then to display these statistics.

1. Recompute the WCET and generates statistics.

   ```
   > owcet -s lpc2138 bs.elf --stats -S
   WCET[main] = 441 cycles
   Total Execution Count:  avg=4.5, max=11, min=0
   Total Execution Time: total=441,  avg=44.1, max=170, min=0
   ```

   In addition to the WCET, we get information about the time of the blocks composing the program and about the number of execution of these blocks in the WCET. These statistics means that:

   - The average, the maximum and the minimum of execution count for WCET per blocks is, respectively, 4.5, 11 and 0.
   - The total, the average, the maximum and the minimum execution time for WCET per blocks is, respectively, 441, 44.1, 170 and 0.

2. To get more details – for instance, the statistics per source line, generate the sources decorated with statistics:

   ```
   > otawa-stat.py -S main -a
   ```

   Where `main` is the name of the top-level function of thet task.

3. Open the obtained decorated sources and observe the obtained pages.

   ```
   > xdg-open main-otawa/src/index.html
   ```

   - What are the most costly source lines in execution time?
   - What are the most executed source lines?

   This source-based statistics display may flatten the dynamic execution of the program and or distort the temporal behaviour of the code. For even more details, one can use the CFG statistics view:

4. The statistics-decorated CFG can be obtained with the command:

   ```
   > otawa-stat.py main -G -S -a
   ```

5. Then they can be opened with the command:

   ```
   > otawa-xdot.py main-otawa/ipet-total_count-cfg/index.dot &
   > otawa-xdot.py main-otawa/ipet-total_time-cfg/index.dot &
   ```

   From `otawa-xdot.py`, you are able to navigate in the CFG and to look in called function CFG by clicking in the *call* nodes.

   - Which blocks are the most costly in execution time – considering all functions of the program?
   - Which blocks are the most frequently executed in the WCET?
   - Are these blocks matching the source lines?
   - Can you determine the path of the WCET?

1.3 What really happens?

This section presents what happens inside **OTAWA** under the hood. Just re-run the WCET computation with the additional option `--log proc`: it displays the details of performed analyses.

```
> owcet -s lpc2138 bs.elf --log proc
```

In the output, each line represents the execution of an anslysis:

```
RUNNING: otawa::dfa::InitialStateBuilder (1.0.0)
RUNNING: otawa::util::FlowFactLoader (1.4.0)
RUNNING: otawa::LabelSetter (1.0.0)
RUNNING: otawa::view::Maker (1.0.0)
RUNNING: otawa::CFGCollector (2.1.0)
RUNNING: otawa::LoopReductor (2.0.1)
RUNNING: otawa::Virtualizer (2.0.0)
RUNNING: otawa::MemoryProcessor (1.0.0)
RUNNING: otawa::lpc2138::AbsMAMBlockBuilder (2.0.0)
RUNNING: otawa::Dominance (1.2.0)
RUNNING: otawa::LoopInfoBuilder (2.0.0)
RUNNING: otawa::lpc2138::CATMAMBuilder (2.0.0)
RUNNING: otawa::ProcessorProcessor (1.0.0)
RUNNING: otawa::lpc2138::MAMEventBuilder (1.0.0)
RUNNING: otawa::ipet::ILPSystemGetter (1.1.0)
RUNNING: otawa::ipet::VarAssignment (1.0.0)
RUNNING: otawa::ExtendedLoopBuilder (1.0.0)
RUNNING: otawa::CFGChecker (1.0.0)
RUNNING: otawa::ipet::FlowFactLoader (2.0.0)
RUNNING: otawa::Weighter (1.0.0)
RUNNING: otawa::etime::StandardEventBuilder (1.0.0)
RUNNING: otawa::lpc2138::BBTime (2.0.0)
RUNNING: otawa::ipet::BasicConstraintsBuilder (1.0.0)
RUNNING: otawa::ipet::FlowFactConstraintBuilder (1.1.0)
RUNNING: otawa::ipet::WCETComputation (1.1.0)
```

The most commonly used analyses are:

`util::FlowFactLoader` – loads the loop bounds,

`CFGColelector` – build the CFGs from the binary code,

`MemoryProcessor` – load the memory description,

`LoopInfoBuilder` – identify loops in the CFG,

`ProcessorProcessor` – load the processor configuration,

`ipet::FlowFactLoader` – assign loop bounds to BBs,

`ipet::BasicConstraintBuilder` – build control flow constraints,

`ipet::FlowFactConstraintBuilder` – build constraints for loop bounds,

`ipet::WCETComputation` – computes the WCET.

Other analyses are specific to our current architecture target:

`lpc2138::CATMAMBuilder` – perform the analysis of prefetch buffers of the flash memory (where is stored the program),

`lpc2138::BBTime` – computes the execution time of each block.

2 Playing with loop bounds and oRange

Directory: `labwork1/crc`
Sources: `crc.c`

`crc` is a small application performing several CRC computations.

This exercise propose a bit more complex program containing several loops. You will have to determine the loop bounds first by hand, then automatically with oRange and observe the differences.

1. Build the executable.

2. Using `mkff`, generate the file "crc.ff".

3. Fill the missing loop bounds ✎ of "crc.ff" using the sources.
   Beware: the bound of one loop of `icrc` depends on the parameters!

4. Compute the WCET ✎ with statistics generation.

5. Generates the CFG view of the statistics.

6. Open the CFG view for statistics `ipet-total_count` and navigate to the function `icrc`. Find the BB (A) corresponding to line `crc.c:93` that represents the head of the loop which body is the block (B) at line `crc.c:94`. Why is the block (A) executed once more than block (B)? ✎

7. In the CFG view, the "main" function contains 2 calls to `icrc`. If you click on the two calling blocks, you get two different CFG. Why? ✎

8. Close `otawa-xdot` window and record the WCET (it is called the *initial* WCET). Remove the file `crc.ff`.

9. As loop bounds are sometimes tricky to obtain, we use now the tool oRange to compute the WCET for us. Type the command:

   ```
   > orange crc.c main -o crc.ffx
   ```

10. This creates a file in XML named `crc.ffx`. Open it with your preferred text editor and observe how the loop bounds are provided, taking into account the subprogram call chains leading to a particular loop. What are now the bound(s) of the loop at `crc.c:93`? ✎

11. Re-compute the WCET with the oRange loop bounds. The new WCET ✎ is lower than the *initial* WCET. Can you explain this using the CFG statistics view and the oRange loop bound file? ✎

3 Playing with loop bounds and the use of total

Directory: `labwork1/bubble`
Sources: `bubble.c`

`bubble` is a tiny application performing the sort of an integer array using the bubble algorithm.

This exercise illustrate the fact, that sometimes, the maximum loop bounds is not enough to produce a tight WCET, specially when the loop bound is dependent on the context. We will work on a pathological case where the calculation of the total number of iterations allows to reduce significantly the overestimation of the WCET.

Notice the difference between two types of bounds:

maximum bound Represents the maximum number of iterations of a loop for each start of the loop.

total bound Represents the total number of iterations of a loop over all the execution of the program.

1. Build the executable.
2. Using `mkff`, generate the file "bubble.ff".
3. Fill the missing loop bounds ✎ of "bubble.ff" using the sources.
4. Compute the WCET ✎ with statistics generation.
5. Generates the CFG view of the statistics.
6. Using the CFG view, lookup and record the execution count ✎ of the BB heading the loop at line 15 in `bubble.c`.
7. From the source, compute the total execution count ✎ of the loop at line 15 in `bubble.c`.
8. What do you observe? Why? ✎
9. To fix the problem, change the loop bound in `bubble.ff` to add, after "`max` N", the syntax "`total` M" with M being the total execution count of the loop.
10. Recompute the WCET ✎ with statistics and observe the reduction of the WCET.

4 Application with several tasks

Directory: `labwork1/helico`
Sources: `helico.c`

`helico.c` is an application driving a quadcopter Unmaned Aerial Vehicle (UAV). It performs a small mission: taking off, keeping stable and landing. This is an actual real-time application. It consists in several real-time tasks that are sequenced in an endless loop contained in `main`() with a period of 1ms.

Therefore, the WCET for the `main`() function cannot be computed but analysing the content of `main`, one can observe there are at least 3 tasks:

- `updateADC`()
- `action`()
- `doPWM`()

To check the schedulability of our real-time application, we have to compute the WCET of tasks that are implemented by these functions.

1. Build the application.
2. Compute the WCET ✎ of `doPWM`() with the command (it does not contain any loop):

   ```
   > owcet -s lpc2138 helico.elf doPWM
   ```

3. As `action`() task contains a loop, first generate and fix a flow fact file for `action`() ✎:

   ```
   > mkff helico.elf action > action.ff
   ```

4. Then compute the WCET ✎ of `action`() with the flow fact file `action.ff`:

   ```
   owcet -s lpc2138 helico.elf action -f action.ff
   ```

5. Do the same with the task `updateADC`() using oRange to compute the WCET ✎.
6. Considering that the management code of the `main` endless loop counts less than 50 cycles: what is the total execution time ✎ of one iteration of this loop?
7. Considering that the `main` endless loop period is 1 ms, is it enough to use an LPC2138 with a frequency of 16 MHz ✎? Computes the smallest possible frequency ✎ to execute in time the `main` endless loop.

5 Enhancing the WCET

Directory: `labwork1/helico`
Sources: `helico.c`

This exercice is the follow-up of the previous exercice, concerning the drive software of a quadcopter application. We will dig down inside the software in order to tighten the WCET of the `main` endless loop.

1. Considering that the `main` endless loop has a period of 1 ms, look inside the function `doPMW`() to find what is the real period✎ of function `updatePWM`() that performs the real work of `doPWM`().

2. Compute the WCET of the following functions using flow fact files computed by oRange:

   - `doGyroChannel`() ✎
   - `doAROMXChannel`() ✎
   - `doAROMYChannel`() ✎
   - `doAROMZChannel`() ✎

   What do you observe about these WCETs✎?

3. Taking into account (a) that a `main` endless loop iteration performs four calls to `updateADC`() with the four possible values for `currentChannel` and (b) that in function `updateADC`(), only one of the functions above is called, could you approximate the WCET of `updateADC`() over the four calls in the sub-loop of `main`✎?

4. Using the previous approximation, compute the approximated total WCET✎ for on iteration of the `main` endless loop iteration? Is there a difference with the WCET of the previous exercise✎?

5. Rewrite the `helico` application in order to avoid the overestimation observed in the previous question and compute the new WCET✎.

6. Re-compute now the minimal frequency✎ required for a processor to run this application.

6 Complex control flow

Directory: `labwork1/control`
Sources: `control.c`
This exercise shows that, even if the control flow of a program is too complex to be analyzed by **OTAWA**, it is still possible to provide by hand the required control flow information.

1. Build the `control` application.

2. Try to compute the WCET of function `main`. You should get something like:

```
WARNING: otawa::CFGChecker 1.0.0:CFG _exit is not connected
        (this may be due to infinite or unresolved branches).
ERROR: CFG checking has show anomalies (see above for details).
```

3. To understand what happens, it is useful to look to the CFG of the application produced by the command:

```
> dumpcfg -Mds control.elf
```

   To view the CFG, the command is:

```
> otawa-xdot.py main-otawa/cfg/index.dot &
```

   You have to remark two things:

   a) The call of the function pointer at line 34 has not been resolved by **OTAWA**✎.

   b) The CFG of the function `_exit`() is disconnect: this comes from the last instruction of `_exit`(), `SWI`✎, that performs a system call to the OS at end of program `control`.

   We have to help **OTAWA** to manage these issues.

4. First, generate the flow fact file for `control` and store it in file `control.ff`. Edit this file.

   It contains new commands representing the issues discovered in the previous question.

   a) The fst command is `multicall` and represents the function pointer call: you have to replace the "?" with the comma-separated list of quoted names of the functions✎ that may be called (look to the sources).

   b) The second entry concernes the `SWI` instruction of `_exit`() function. `mkff` proposes to consider this function as either a non-returning instruction, or a call to multiple functions. Just remove the bad line✎.

5. Rebuild the CFG and check that the new CFG is now consistent, that is, connected without any unknown call or branch.

6. Now, you can compute the WCET✎.