Exercise Sheet 1: Using OTAWA framework

By H. Cassé.

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

If you want to work on your laptop, you can use this to get a script in order to compile and install otawa (but it will take a while). This creates a directory named `otawa` and the commands are in directory `otawa/bin`. Notice that the directory `otawa` can be moved around without breaking the insyallation.

On our desktop, nothing needs to be done: OTAWA commands are already on the path.

Assignment

This document contains exercice with empty fields that you have to fulfill and then you have to deposit this file on the Moodle repository. The question you have to answer are marked with symbol [=>]. When you have type free explanation, they have to be framed between [<<] and [>>].

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 perform 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.

Calculating the WCET

PROFI

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 OS 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:

```sh
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 + 0 xb4 (000101 b4 ).
WARNING : otawa :: ipet :: FlowFactLoader 2.0.0: in the context [ FUN
(000101 dc ) , CALL (000101 e8 ) , FUN (00010100)]
ERROR : otawa :: Weighter (1.0.0): cannot compute weight for loop at BB
2 (000101 b4 )
```

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 question mark `?` with syntax `max` *N*, with *N* being the loop bound, actually 5. The fixed `bs.ff` is displayed below:

```
checksum " bs.elf " 0xc39d57a1 ;
// Function binary_search (bs.c:82)
loop " binary_search " + 0xb4 max 5 ; // 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 it depends on the used compiler version).

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

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 these 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 -W
   WCET [ main ] = 441 cycles
   ```

2. You can observe that a directory named `bs-otawa/main` (executable `bs.elf`, task `main`) has been created. It contains all details about the computed WCET and can be viewed with the utility `obviews`:

   ```
   $ obviews.py bs.elf
   ```

`obviews` displays a window (or open it in your preferred browser) where all aspects of WCET computation are graphically displayed:

- On the left, you can click on the functions composing the program and have the CFG displayed.
- Also on the left, you have the list of sources and you can display them.
- In addition, you can colorize CFGs and sources according to the WCET contribution by selecting the displayed stats (combo `No stat.`).
- With the View button, you can configure what is displayed in the CFG nodes (source, assembly, etc). Click on Done when done.

PROFI

Basically, there is two types of statistics:

- *Total Execution Count* - number of executions in the WCET.
- *Total Execution Time* - cumulated time of execution in the WCET (maybe the most interesting).

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
RUNNING: otawa::script::Script (1.0.0)
INFO: plugged otawa/lpc2138
(/home/casse/Enseignement/WCET/otawa/lib/otawa/otawa/lpc2138.so)
RUNNING: otawa::dfa::InitialStateBuilder (1.0.0)
RUNNING: otawa::FlowFactLoader (1.4.0)
RUNNING: otawa::LabelSetter (1.0.0)
RUNNING: otawa::TaskInfoService (1.0.0)
RUNNING: otawa::view::ViewBaseImpl (1.0.0)
RUNNING: otawa::CFGCollector (2.2.0)
RUNNING: otawa::LoopReductor (2.0.1)
RUNNING: otawa::Virtualizer (2.1.0)
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::lpc2138::EventBuilder (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::lpc2138::BBTime (2.0.0)
RUNNING: otawa::ipet::BasicConstraintsBuilder (1.0.0)
RUNNING: otawa::ipet::FlowFactConstraintBuilder (1.1.0)
RUNNING: otawa::ipet::FlowFactConflictConstraintBuilder (1.1.0)
RUNNING: otawa::ipet::WCETComputation (1.1.1)
```
~~~~

PROFI

The most commonly used analyses are:

- `util::FlowFactLoader` - loads the loop bounds,
- `CFGCollector` - 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.

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!

```
[=>] Loop bounds:

[<<]
    source "icrc"  + line 104 bound max 8
    source "icrc"  + line 93 bound max 128
    source "icrc1" + line 73 bound max 40

[>>]
```

4. Compute the WCET with statistics generation.

   [=>] WCET = 291025 cycles

PROFI

5. Open `obviews` on the program and navigate to the CFG `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)?

   [=>] Block (A) address: 0x5c8 Execution count: 257

   [=>] Block (B) address: 0x520 Execution count: 256

   [=>] Why A is executed once more than (B)?

   [<<] Le bloc (A) correspond au test de la condition de la boucle (en-tête). Le bloc (B) est le corps de la boucle. (A) s'exécute une fois de plus que (B) car il est exécuté une dernière fois à la fin pour vérifier que la condition est fausse et pouvoir sortir de la boucle.

   [>>]

6. 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. Observe the CFG call path at the top just above the left view, one is called from line 131 and the other one from line 134.

   [=>] Why **OTAWA** separated the CFGs?

   [<<] otawa separe les graphes car la fonction `icrc` est appelée depuis deux endroits différents (lignes 131 et 134). Cela permet d'analyser chaque appel séparément (contextes differents) et d'avoir des bornes de boucles precise pour chaque cas, au lieu de melanger les deux.

   [>>]

7. Record the WCET (it is called the initial WCET). Remove the file `crc.ff`.

   [=>] initial WCET = 291025 cycles

8. 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
   ```

9. 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`?

   [=>] bound of loop at `crc.c:93` = 256

10. Re-compute the WCET with the *oRange* loop bounds (**OTAWA** will automatically use the file `crc.ffx`). The new WCET is lower than the initial WCET. Can you explain this using `obviews.py` and the *oRange* loop bound file?

    [=>] new WCET = 81795 cycles

    PROFI

    [=>] Why is there a difference?

    [<<] La difference vient de l'analyse au contexte d'oRange. Au debut, on avait forcé manuellement le pire cas (128 iterations) pour tous les appels.oRange a détecté que lors du deuxième appel (ligne 134), la condition de boucle fait qu'on y entre pas (0 itération). Comme on évite toute cette exécution inutile dans le calcul, le WCET chute drastiquement.

    [>>]

Playing with loop bounds and 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 all over the execution of the program.

1. Build the executable.
2. Using `mkff`, generate the file `bubble.ff`.
3. Fill the missing loop bound of `bubble.ff` using the sources (not *oRange*).

   [=>] loop bounds

   [>>] source bubbleSort + 0x108 line 12 bound 7 source bubbleSort + 0xe0 line 15 bound 7 [>>]
4. Compute the WCET with statistics generation.

   [=>] initial WCET = 3425 cycles
5. Using `obviews.py`, lookup and record the execution count of the BB heading the loop at line 15 in `bubble.c`.

   [=>] execution count of loop at `bubble.c:15` = 64
6. From the source, compute the real total execution count of the loop at line 15 in `bubble.c` (taking into account the value of the parent loop at line 12).

   [=>] total execution count computed by hand = 28

PROFI

7. What do you observe? Why?

   [=>]

   [<<] OTAWA considère que la boucle interne tourne tout le temps 7 fois (le max) à chaque passage.Mais en vrai, elle diminue à chaque tour (7, 6, 5...) parce qu'elle dépend de `i`. Du coup, OTAWA compte 56 itérations au total alors qu'il n'y en a que 28 en réalité. [>>]
8. 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 computed in the previous question.
9. Recompute the WCET with statistics and observe the reduction of the WCET.

   [=>] new WCET = 1857 cycles

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 1 ms.

Therefore, the WCET for the `main()` function cannot be computed but by analysing the content of `main`, one can observe that the main work is done by the function `stabilize()` that is composed of 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
   ```

   [=>] `doPWM()` WCET = 156 cycles

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

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

PROFI

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

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

   [=>] `action()` WCET = 314 cycles

5. Do the same with the task `updateADC()` using *oRange* to compute the WCET

   [=>] `updateADC()` WCET = 1917 cycles

Function `stabilize()` is the main content of the endless loop implented in main. To test whether the real-time system meets its temporal constraint, we have to check that the execution of the loop (in fact that `stabilize()` takes less than 1ms).

6. How many times the loop in `stabilize()` iterates?

   [=>] `stabbilize()` loop bound = 4

7. Considering that the management code of the loop of `stabilize()` counts less than 50 cycles: what is the total execution time of `stabilize()`?

   [=>] total execution time of `stablize()` = 9598 cycles

8. How many cycles are available for one iteration of the stabilize function with an LPC2138 at a frequency of 8 MHz (i.e. cycles for a period of 1ms) ? From the WCET computed just above for stabilize, is it enough to use an LPC2138 to run `helico` code? Computes the smallest possible processor frequency to execute in time the function stabilize.

   [=>] 1ms on the LPC2138 = 8000 cycles [=>] is LPC2138 powerful enough to run `helico` ☐ yes / [x] no [=>] smallest possible frequency for LPC2138 : 9 598 000 Hz

Enhancing the WCET

Directory: `labwork1/helico`

Sources: `helico.c`

This exercice is the follow-up of the previous exercice, concerning the autopilot 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 1ms, look inside the function `doPMW()` to find what is the real period of function `updatePWM()` that performs the real work of `doPWM()`.

   [=>] real period = 10ms

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

- `doAROMXChannel()`

  [=>] WCET = 177 cycles

- `doAROMYChannel()`

  [=>] WCET = 163 cycles

- `doAROMZChannel()`

  [=>] WCET = 162 cycles

- `doGyroChannel()`

  [=>] WCET = 1828 cycles

  What do you observe about these WCETs?

  [=>]

PROFI

[<<] On observe une énorme différence. `doGyroChannel` est beaucoup plus coûteux (1828 cycles) que les trois autres (~170 cycles).Cela s'explique car `doGyro` exécute toujours son filtre, alors que les canaux AROM ne font le calcul lourd que si `heliState == HOVER` (ce qui n'est pas le cas ici). [>>]

3. Taking into account (a) that `stabilize()` loop 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 at each call, could you improve the WCET of `updateADC()` over the four calls ?

   [=>]

   [<<] Oui ! pour calculer le WCET de `stabilize`, on fait "4 fois le pire cas de updateADC". Le pire cas est `doGyro` (1917 cycles). otawa compte donc 4 * 1917 = 7668 cycles pour la partie ADC. on sait qu'on appelle une fois Gyro, une fois X, une fois Y et une fois Z. On devrait donc additionner les WCET individuels (1828 + 177 + 163 + 162 = 2330) au lieu de multiplier le pire par 4.

   [>>]

4. Using the previous `updateADC()` approximation, compute the approximated total WCET for one call of `stabilize()`? Is there a difference with the WCET of the previous exercise?

   [=>] approximated WCET = 4260 cycles

   [=>] difference

   [<<] Le WCET précédent était de 9598 cycles. Le nouveau est de 4260 cycles. C'est une réduction de plus de 50% ! Cela rend le système planifiable sur le processeur (car 4260 < 8000).

   [>>]

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

   [=>] new WCET = 4639 cycles

PROFI

6. Re-compute now the minimal processor frequency required for a processor to run this application. Is the LPC2138 at 8MHz is now enough?

   [=>] new minimal frequency = 4 639 000 Hz

   [✓ yes] / [ ] no

7. Deposit on Moodle's repository your new version of `helico.c`.

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 - Wds control.elf
```

And then explore it with `obviews.py`:

```
$ obviews . py control.elf
```

You have to remark two things:

a. The call of the function pointer at line 32 has not been resolved by **OTAWA**.

[=>] BB address containing this call = 0x4ac

b. The CFG of the function `exit()` is disconnected: this comes from the last instruction of `exit()`, `SWI` that performs a system call to the OS at end of program to exit (*enable disassembly view*).

PROFI

```
[=>] SWI address = 0x4e0
```

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

```
[=>] called functions = one, zero
```

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

```
[=>] Which line do you keep?

[=>] [x] non-return function

[=>] [ ] multi-call
```

5. Re-generate the CFG with `dumpcfg` 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.

   [=>] WCET = 49 cycles

er this function as either a non-returning instruction, or a call to multiple functions. Just remove the bad line.

```
[=>] Which line do you keep?

[=>] [x] non-return function

[=>] [ ] multi-call
```

5. Re-generate the CFG with `dumpcfg` 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.

   [=>] WCET = 49 cycles