STREC - TD-IPET Énoncé

General Instructions

- You can download the source files for the exercise from: `https://strec.wp.mines-telecom.fr/TD-IPET/`

Today's exercises will be based on the `Otawa` Worst-Case Execution Time[1] analysis tool, the `Patmos` processor[2], and a simple flight control software (`Heli`). The following three sections are intended to provide a brief introduction, the actual exercises then follow below.

Heli

Heli is a simple flight control software for a helicopter drone. Software and hardware were initially developed in 2006 by Idan Beck and Rohit Gupta. Both were students at Cornell University at that time. For details see their report.

The software was then adapted for the Worst-Case Execution Time Challenge 2014. More information is available on the tool challenge's website.

Patmos Processor

`Patmos` is a newly designed processor aiming at time-predictability, which means that several aspects of the processor are designed specifically for the use in (hard) real-time systems. We will not have the time to actually make use of most of its features in this class though.

For the following exercises it suffices to know that the processor is based on a 5-stage pipeline that executes instruction in-order. Instructions are assumed to take a single cycle to execute, memory accesses are assumed to be for free (we will cover aspects related to caches and memory accesses next time). `Patmos`' instruction set follows typical RISC (Reduced Instruction Set Computer) conventions: (1) almost all instructions operate on registers, while (2) dedicated load/store instructions allow to access memory. The processor offers 32 general-purpose registers (32 bit each) as well as 8 predicate registers (1 bit each) and 16 special-purpose registers (32 bit each – but we won't actually use them). A more detailed overview is provided in the "Patmos Handbook".

You will need to disassemble the machine code of the provided flight control software during the exercises, this can be done using the tool `patmos-llvm-objdump` as follows:

[1]`http://www.otawa.fr/`
[2]`http://patmos.compute.dtu.dk/`

```
brandner@kairon:~/> patmos-llvm-objdump -d -s heli  2>&1 | less
heli:   file format ELF32-patmos

Disassembly of section .text:
.text:
   20080:       00 00 00 90                         li     $r0 = 144

.LBB0_0:
_start:
   20084:       87 c2 00 00 f0 00 00 00             li     $r1 = -268435456
   2008c:       02 82 10 80                         lwl    $r1 = [$r1]
   20090:       00 40 00 00                         nop
...
```

The output shows, for each address (left-most column), the disassembled instructions (right-most column) as well as the binary representation of the instructions as hexadecimal numbers (middle).

Otawa

Otawa is an open-source WCET analysis tool that closely follows the analysis approach described in the lecture. In particular, the tool performs (1) a pipeline analysis, before (2) constructing an integer linear program (ILP). The ILP corresponds to the IPET approach covered in the lecture. The IPET equations are finally solved using a generic ILP solver (`lp_solve` in our case).

In order to perform an analysis three input files are required:

- Executable:
  An executable file (`ELF`) containing the machine code of the program to analyze (`heli` in our case).
- Flow facts:
  Additional information, such as loop bounds, are provided in a separate flow-fact file (`heli.ff`) based on the `F4` file format.
- Platform configuration:
  Finally, an XML-based platform configuration file (`patmos_wcet.osx`) is provided. It describes the processor features, memory characteristics, and analysis steps that need to be performed.

The `Heli` application, for instance, can be analyzed using the following command:

```
brandner@kairon:~/> owcet -s patmos_wcet.osx -f heli.ff heli
warning: reverting to default arch plugin
warning: reverting to default sys plugin
INFO: plugged tcrest/patmos (lib/otawa/proc/tcrest/patmos_wcet.so)
INFO: plugged otawa/display (lib/libodisplay.so),
WCET[main] = 1061372 cycles
```

The command specifies the platform configuration (`-s patmos_wcet.osx`), followed by the flow-fact file (`-f heli.ff`), and the name of the executable (`heli`). The last line of the output indicates the computed WCET of the program in processor execution cycles ($1\,061\,372$).

1 Control-Flow Reconstruction (20 minutes)

Aims: *Understand the reconstruction of the program's control-flow from machine code.*

Download the archive `TD-IPET-STREC.tar.gz` for today's exercises from the course website as indicated above. After decompressing the archive you will find several files:

```
./heli
./heli.ff
./Makefile
./patmos_wcet
./patmos_wcet/caches.xml
./patmos_wcet/memory.xml
./patmos_wcet/pipeline.xml
./patmos_wcet.osx
./src/heli.c
./src/io.h
```

- Invoke `make` in order to trigger a first WCET analysis run.
- Check the output of the analysis tool.

  What is the WCET reported by the tool for the function `main`?
- Along with the output on the console, the analysis also generated plenty of intermediate files (see the directory `./out/`). Open the file `./out/fixFilter.dot` using `xdot`. This should display the control-flow graph of the `fixFilter` function. At the same time open the source code `./src/heli.c`, search for the function, and try to match the control-flow graph with the C source code.

  Which basic blocks correspond to the `for` loop? .
- Open the flow fact file (`heli.ff`). Try to find out what the first line (**`multibranch`**) does, e.g., by removing on of the entries after the keyword **`to`**.

  Hint: Use the `xdot` too to visualize the control-flow graph, search for the basic block corresponding to the address indicated before the **`to`** keyword and then check if the control-flow graph in `./out/processSensorData.dot` changes when you modify the flow fact file.

  What is the purpose of the `multibranch` statement? Why is `Otawa` not able to find this information on its own? Compare this with the handling of *normal* branches (e.g., at the end of basic block 1)? Which C statements correspond to this instruction?

- **Bonus Question:** Disassemble the `Heli` executable and verify that the **`multibranch`** statement is correct.

  Hint: The branch address is loaded from an array that is stored in memory at address 236524 (`0x39bec`). Use `patmos-llvm-objdump` in order to find the table.

2 Loop Bounds (30 minutes)

Aims: *Understand loop bound specifications and calling contexts.*

Attention: Do not forget to undo any modifications to the flow fact file that you may have done during the previous exercise.

- Again open the source code of `Heli` (`./src/heli.c`) and have a closer look at the `for` loop in the function `fixFilter`. Find the **`loop`** statement in the flow fact file corresponding to the loop.

  Hint: Check all the locations where the function is called in the source code.

  Which loop bound is currently indicated in the flow fact file? Determine an improved, but still safe, loop bound for this loop?

- Modify the flow fact file according to your findings from above and rerun the analysis (`make`).

  What is the new WCET bound computed by `Otawa`?

- The current bound is obviously pessimistic, since it is easy to determine the precise loop bound depending on the calling context, i.e., whenever `fixFilter` is called from `calibrateGyro` the loop bound is known to be $2^5 = 32$ instead of 256. We thus would like to activate calling contexts during the analysis.

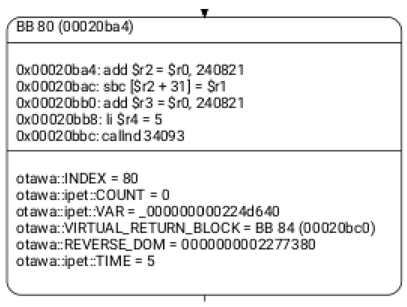
  This can be done by modifying the platform configuration file. Open the configuration file (`./patmos_wcet.osx`) and uncomment the line following the marker `TODO` (l. 17). Then rerun the analysis and visualize the control-flow graph of the function `main` (`./out/main.dot`). The control-flow graph appears much larger?!?

  Hint: Search for the name `fixFilter` in the control flow graph. You can use the input field in the toolbar for this – `xdot` highlights all matching nodes in red.

  Explain the difference between the *small* original graph for `main` and the *large* new graph?

- Now replace the pessimistic loop bound by precise loop bounds for each call site of `fixFilter` that you can find in the control-flow graph.

  Figure 1, for instance, illustrates such a call site. Note the instruction **`li`** `$r4 = 5` before the actual call instruction. Register `r4` is used to pass the value 5 to the function's `size`



```
BB 80 (00020ba4)

0x00020ba4: add $r2 = $r0, 240821
0x00020bac: sbc [$r2 + 31] = $r1
0x00020bb0: add $r3 = $r0, 240821
0x00020bb8: li $r4 = 5
0x00020bbc: callnd 34093

otawa::INDEX = 80
otawa::ipet::COUNT = 0
otawa::ipet::VAR = _000000000224d640
otawa::VIRTUAL_RETURN_BLOCK = BB 84 (00020bc0)
otawa::REVERSE_DOM = 0000000002277380
otawa::ipet::TIME = 5
```

Figure 1: Call to the function `fixFilter` in the control-flow graph exported by `Otawa`.

parameter. Thus the number of loop iterations for the particular call site shown in the figure can be bounded by 32.

It remains to provide this information to `Otawa` in the form of a new **`loop`** statement using the following format:

```
loop <loop address> <loop bound> in @<call address>;
```

The loop address remains unchanged (you can copy it from the existing **`loop`** statement, whereas the loop bound can easily be derived from the **`li`** that you will find before each call site. The address of the call, finally, is specified by an *at sign* (`@`) followed by the address of the call instruction (`0x20bbc` in Figure 1).

Hint: Do not forget to remove the original **`loop`** statement in the flow fact file.

Provide precise loop bounds, following the above example, for all 8 call sites of the function `fixFilter`. What is the WCET that you obtain after this improvement?

3 Pipeline Analysis (15 minutes)

Aims: *See a basic pipeline at work.*

The pipeline analysis in `Otawa` is based on so-called `ExeGraphs`, which represent the execution of a sequence of instructions on a pipelined processor in a compact way. The graph contains a node for each instruction and each pipeline stage. This results in a rectangular structure where the height corresponds to the length of the instruction sequence and the width to the number of pipeline stages of the processor. At least one such graph is constructed for each basic block in the control-flow graph.

In order to improve precision, `Otawa` extends the instruction sequence by including some instructions of the preceding basic block. If multiple predecessors exist multiple graphs are constructed.

Once all `ExeGraphs` are constructed for a basic block, the time required to execute the basic block is computed by considering the difference between the start date of the last pipeline stage of the last instruction of the predecessor block as well as the start date of the last pipeline stage of the last instruction in the sequence.

The maximum value over all the differences of all its `ExeGraphs` is then considered the worst-case execution time of the basic block. This value is later used during the IPET phase (see the next exercise).

- The current model of the `Patmos` processor assumes that every instruction takes precisely one cycle to execute.

  Search for basic block 21 and compute the number of cycles it takes to execute this basic block.

- Open the `ExeGraphs` of basic block 21. This basic block represents the `for` loop of the function `fixFilter` and thus has two predecessors (the basic block itself when executing the loop, and another basic block to enter the loop). The two graphs can be found in the files `./out/b19-ctxt21-case0.dot` and `./out/b21-ctxt21-case0.dot`.

  Hint: Nodes of instructions that belong to the current basic block are represented in blue, while instructions belonging to the predecessor are represented in black.

  Verify that the execution time of the basic block computed by `Otawa` matches the expected 7 cycles.

4 Implicit Path Enumeration (40 minutes)

Aims: *Understand the relationship between the control-flow graph, loop bounds, and pipeline analysis as well as the equations of the Implicit Path Enumeration Technique (IPET).*

The previously constructed control-flow graph (see Exercise 1) is annotated with the user-provided flow facts and loop bounds (Exercise 2) as well as the local execution times of basic blocks (Exercise 3). The final step of WCET analysis is then to construct linear equations according to the Implicit Path Enumeration Technique (see the lecture).

You can have a look at the constructed equations by opening the LP-file generated by `Otawa` (`./out/ipet.lp`). The variable names in the equations represent the basic blocks and the edges of the control-flow graph. The name of a variable representing a basic block u is given by `x<u>_main`. For instance, a variable `x21_main` is introduced for basic block 21. The naming of edges follows a similar scheme: for an edge (u, v) in the control-flow graph a variable `e<u>_<v>_main` is introduced. For instance, the variable `e21_21_main` is introduced for the loop-edge of basic block 21 that leads to the block itself. The variables are used to encode to structural (flow) constraints, as well as the objective function. Note that the weight of an edge (u, v) corresponds to the local execution time of the destination block v.

- Open the LP-file generated from an analysis run. Try to find all variables related to basic block 21 (corresponding to the loop in the first invocation of the function `fixFilter`).

  Determine the weights associated with the corresponding edges in the objective function. Then, find all equations related to loop bounds. Finally, find all equations representing structural constraints (Kirchhoff's law).

- Now verify that the solution to the equations in the LP-file actually corresponds to the expected value $33\,886$ (from Exercise 2). Supply the generated LP-file as input to the tool `lp_solve` and examine its output.

  Hint: Invoke `lp_solve` with the option `-S1` in order to reduce the amount of output generated.

- The solution computed by `lp_solve` is actually back-annotated to the control-flow graph. Reexamine the generated graph using `xdot`. Search for the string `otawa::ipet::COUNT`. It indicates the value of the flow variable (given by `otawa::ipet::VAR`) computed by `lp_solve`.

  Have a close look at the control-flow graph for the function `runFlightPlan` (starting at basic block 50) as well as the corresponding C code. You will quickly notice that the `if` statements in the C code are mutually exclusive.

  Hint: Check basic blocks 50, 51, 53, 55, 58, and 61.

  Verify the flow variables (`otawa::ipet::COUNT`) associated with the function's control-flow graph. Did the tool succeed to exploit the fact that the `if` statements are mutually exclusive? Justify your answer.

- Unfortunately the flow fact parser of `Otawa` does not support the required features to express mutually exclusive `if` statements. We will thus directly modify the LP-file.

  Hint: `lp_solve` only accepts linear equations where the right hand side is a constant. Furthermore the operators in the equations are limited to <, <=, and =. You thus may have to adapt your equations to respect these limitations.

Propose an equation that allows to express the mutual exclusivity of the `if` statements. Add the equation to the LP-file and recompute a new solution with `lp_solve`. The obtained WCET should be $33\,534$ cycles.

- Next have a closer look at the `switch` statement in function `processSensorData`. You will again quickly notice that the variable `currentChannel` represents a sort of a state machine that repeatedly cycles through the following states: `GYRO_CHANNEL`, `AROMX_CHANNEL`, `AROMY_CHANNEL`, and `AROMZ_CHANNEL`. Note that the state machine wraps around when the last state is reached.

  Hint: The `case` statements representing the states `GYRO_CHANNEL`, `AROMX_CHANNEL`, `AROMY_CHANNEL`, and `AROMZ_CHANNEL` correspond to basic blocks 68, 69, 70, and 71 respectively.

  Propose a set of equations that captures the behavior of the state machine. Again, add these equations to the LP-file and recompute a solution using `lp_solve`. The obtained WCET should now be reduced to $22\,545$ cycles.