Exercise Sheet 2

Programming with OTAWA framework

H. Cassé <casse@irit.fr>

The goal of this exercise is to use **OTAWA** to develop your own analysis for WCET computation. We will use **OTAWA** framework to:

- write an analysis computing time of BB for a particular microprocessor pipeline supporting *ARM* instruction set,
- implement the static analysis to account for the access of the flash memory (where is stored the program) and its *fetch buffer*.

Important The documentation of **OTAWA** classes is available at:
file:/nfs/otawa/site/share/Otawa/autodoc/index.html.

Assignment You have to submit the resulting sources to the corresponding repository on Moodle!

1 OTAWA environment

OTAWA framework is made of several commands and libraries. In our case, all resources are contained in `/nfs/otawa/site`. It is organized in a very usual way:

bin contains commands.

lib contains libraries.

include C++ header files.

share miscellaneous data, scripts, documentation, etc.

To achieve the exercise, you to complete a source file contained in an archive that must be unpacked and compiled in your own directory (also available on Moodle website):

```
> tar xvf /nfs/otawa/labwork2.tgz
> cd labwork2
> cmake . -DOTAWA_CONFIG=/nfs/otawa/site/bin/otawa-config
> make
```

To test your analyses, you have to use the *ELF* binary files of the directory `test`:

bubble.elf Implements a bubble sort.

matmul.elf Implements a matrix multiplication.

This test binaries are delivered with their sources (`.c`) and their flow fact files (`.ff`).

To understand the work of your analysis, it may be useful to have a representation of the CFG of the test programs. A `.dot` output representing the CFG is easily obtained with command:

```
> dumpcfg -Mds BINARY.elf
```

This creates a directory containing `.dot` files for each CFG of the binary. They can be viewed and navigated with command:

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

To do Visualize the CFG of both test programs `bubble.elf` and `matmult.elf`.

2 Computing the execution time of a BB

In this first exercise, we want to implement the time computation for a pipeline with the following instructions timings:

time	type of instruction
4 cycles	multiplication
5 cycles	memory load
2 cycles	memory store
2 cycles	conditional control instruction
1 cycle	other instructions

To implement the block time computation, you have to fulfill the file `labwork.cpp`. This source file (in C++) contains several class definitions but we will focus on class `TimeBuilder`. It extends the class `BBProcessor` that indirectly extends the class `Processor`. In **OTAWA**, all analyses extend this class that provides all details about the execution of an analysis:

- analysis name and version,
- required features (resources required from other analyses),
- provided features (resources provided by this analysis),
- etc

In **OTAWA**, a feature is a collection of annotations. An annotation is any type of information refered by an identifier and hung to some part of a program representation. For example, the annotation `ipet::TIME` refer to the execution time of a BB and is hung to an object of class `BasicBlock`.

The program representation is basically made of a `File` (the `.elf` file) embedded in a `WorkSpace` that provides all facilities to perform analyses on the binary code. A classic representation of a program is as a collection of `CFG`, one `CFG` for each function composing the program. Each `CFG` is composed of a collection of `Block`. A `Block` may be an end block (representing entry and exit points of the CFG) or a BB (class `BasicBlock`). BB are themselves made of a list of machine instructions (class `Inst`).

Using an analysis extending the class `BBProcessor`, you do not have to worry about the CFG structure of the program because the function `processBB()` is called for each block: you have just to fill the body of this function and to hang the `ipet::TIME` annotation to the BBs with the syntax:

```
ipet::TIME(b) = t;
```

With `b` a block and `t` the time of the BB.

An annotation can be read back using the syntax:

```
cout << "time␣=␣" << *ipet::TIME(b) << io::endl;
```

To do

1. Implement the function `processBB`() of class `TimeBuilder` and compile it.
2. Test it with the given test programs.
3. Check your results based on the CFG output produced in the previous section.

The built program is invoked with:

```
> ./labwork BINARY.elf
```

OTAWA API To iterate over the instructions of a basic block, type:

```
for(auto i: *bb)
        // do something with instruction i
```

The following functions are available on instructions:

i->isMul() returns true if `i` is a mutliplication.

i->isLoad() returns true if `i` is a memory load.

i->isStore() returns true if `i` is a memory store.

i->isControl() returns true if `i` is a control flow instruction.

i->isConditional() returns true if `i` is a conditional instruction.

The parent class `Processor` provides the following facilities:

log Output stream representing log information of the analysis (usually displayed on `cerr`).

isVerbose() returns true if the verbose option is set in **OTAWA** configuration. This may be used to provide details about the work of your analysis.

When you run your program, the *verbose* mode is enabled and all details about all performed analyses are printed on the screen. Scroll up the output to find a line named `Starting TimeBuilder` to look to the outputs of your analysis.

Note In reality, computing the execution of a BB for a pipeline is much more complicated requiring to set up *Execution Graph* and taking into account variable times from caches, memories, etc. Yet, we are a bit limited in labwork time.

3 Implementing a static analysis to cope with flash memory access time

The computation performed so far does not take into account the flash memory timings (containing the program memory). The flash memory is divided into pages of 64 bytes and reading a page takes *20 cycles*. To speed up the execution of the program, the flash memory is equiped with a *fetch buffer* containing one page. If the accessed blocks is in the buffer, the accessed time is zero (transparent for the fetch stage).

The objective of the analysis performed in class `FlashAnalysis` is to increase the execution of a block, `ipet::TIME`, with the time devoted to flash memory accesses when a *miss* occurs (when the accessed page is not in the fetch buffer). Depending on the predicted state of the flash *fetch buffer*, an instruction may takes 20 cycles if the page is not already loaded in the buffer (*miss*), or 0 cycles otherwise (*hit*).

This means that we have to implement an abstract interpretation to compute the possibles pages contained in the *fetch buffer* at each point of the program. The components of this abstract interpretation are:

- the abstract state $\mathcal{S}$ of the *fetch buffer* – the address of the page currently in the *fetch buffer* and two special values: `TOP` represents any value and `BOT` represents the undetermined value:
  $\mathcal{S} = Address \cup \{\top, \bot\}$
- the join operation of two abstract states (function `join` of `FlashAnalysis`):
  $\forall a_1, a_2 \in \mathcal{S}, a_1 \sqcup a_2 \in \mathcal{S}$
- the update operation that computes the state of the *fetch buffer* after the execution of an instruction (function `update` of `FlashAnalysis`):
  $update : Inst \times \mathcal{S} \mapsto \mathcal{S}$

Notice that the class `Address` is used in **OTAWA** to represent addresses. It supports a comprehensive set of operations usually applied to addresses and may be provided by most classes like `Inst`, `BasicBock`, etc. To get the integer value of an address, one has to use the function `Address::offset()`.

The analysis has to be implemented in function `processAll()` of `FlashAnalysis`. You have to proceed in two steps:

1. Compute the abstract state of the *fetch buffer* at each exit of the blocks (it is stored using `OUT` annotation). As the CFG may contains loops, a fixpoint computation is required.
2. Use the results of the previous analysis to compute the flash memory access cost of each instruction: 0 cycle if the page is in the buffer, 20 additional cycles if we are not sure wether the page is in the fetch buffer.

The algorithm 1 shows the implementation of our analysis using a working list.

Data: $G = (V, E, \alpha)$ – CFG
V vertices of G
$E \subseteq V \times V$ edges of G
$\alpha \in V$ – entry vertex of G
Result: OUT_v – fetch buffer state after BB $v \in V$
$OUT_\alpha \leftarrow \top$
$todo \leftarrow SUCC(\alpha)$
while $todo \neq \emptyset$ **do**
  $pop\ v\ from\ todo$
  $IN \leftarrow \bigsqcup_{w \in PRED(v)} OUT_w$
  $OUT \leftarrow update(v, IN)$
  if $OUT \neq OUT_v$ **then**
    $OUT_v \leftarrow OUT$
    $todo \leftarrow todo \cup SUCC(v)$
  end
end

Algorithm 1: Point-fix algorithm to compute fetch buffer abstract states.

To do Re-compile and fix mistakes after each question.

1. Implement the join operation in the C++ function `join()` of `FlashAnalysis`.
2. Implement the update operation in the C++ function `update()` of `FlashAnalysis`.
3. Implement the C++ function `input()` of `FlashAnalysis` that computes the input state of a block (join of output states of the predecessors – stored with annotation `OUT`). Notice that, if an annotation `OUT` is not defined for a BB, its default value is $\bot$ that is compliant with our analysis.
4. Implement the C++ function `processBasicBlock`() that takes a BB as parameter and compute the state after the BB stored using the annotation `OUT`. This function returns true if the new output state is different from the former one.
5. Implement the C++ function `processAll()` of `FlashAnalysis` that (a) perform the computation of OUT_v and (b) fix the execution time of blocks according to the states of the buffer.
6. Complete the function `processAll()` to add to the execution time of the block `ipet::TIME` the time penalty of flash accesses: a *miss* of 20 cycles arises each time an instruction is contained in page different from the one in the *fetch buffer*.
7. Compile and test your analysis. Use the CFG output by `dumpcfg` and the displayed log to check your analysis.
8. Modify the analysis to display the *hit rate* and the *miss rate* of the analysis.

9. The calculation of the input state by joining the output states of previous vertices causes a significant overestimation: a lot of flash memory blocks spans over two sequential BB but the first instruction of the second BB is often accounted as missing with a 20-cycle access. To prevent this, compute the flash memory access time along edges for each predecessor and hang this time to the corresponding edge using the property `ipet::TIME`.

OTAWA API To Iterate on the predecessors of a block `v`:

```
for(auto e: v->inEdges())
        work_with_predecessor(e->source());
```

To Iterate on the successors of a block `v`:

```
for(auto e: v->outEdges())
        work_with_successor(e->sink());
```

The following functions may be useful:

page(i) Return the page address of instruction i.

cfgs().entry()->entry() gets the entry block $\alpha \in V$ of the main CFG.

e->source() gets the source block of edge `e`.

e->sink() gets the sink block of edge `e`.

todo.pop() gets next element from vector `todo`.

todo.push(v) adds block `v` to the vector `todo`.

i->address() gets the address of instruction `todo`.

todo.isEmpty() returns true if the vector is empty.

Hint (1) The abstract interpretation proposed in the exercise is relatively simple: either there is exactly one unique page stored in the *fetch buffer* and its value is the address of the page, or it may be several pages and the value is $\top$.

Hint (2) If you are still blocked, you can use the definitions below:

$$\forall s \in \mathcal{S}, \forall i \in Inst, update(s, i) = page(i)$$

With $page(i) : Inst \mapsto Address$ returning the *fetch buffer* page address containing i.

$$\forall s_1, s_2 \in \mathcal{S}, s_1 \sqcup s_2 = \begin{cases} s_1 & if\ s_2 = \bot \ \vee \ s_1 = s_2 \\ s_2 & if\ s_1 = \bot \\ \top & else \end{cases}$$