Test Engine $branch$

Parameters

- $branch\_exec\_limit$ is an upper bound for the number of executions of a single branch instruction;
- $trace\_count\_limit$ is an upper bound for the number of execution traces to be returned.

More information on the parameters is given in the "*Execution Traces Enumeration*" section.

Description

Functioning of the $branch$ test engine includes the following steps:

1. construction of a *branch structure* of an abstract test sequence;
2. enumeration of *execution traces* of the branch structure;
3. concretization of the test sequence for each execution trace:
    a. construction of a *control code*;
    b. construction of an *initialization code*.

Let D be the size of the delay slot for an architecture under scrutiny (e.g., $D = 1$ for MIPS, and $D = 0$ for ARM).

Branch Structure Construction

Step 1 consists in scanning the instructions of the abstract test sequence and constructing the branch structure. A branch structure is a finite and nonempty sequence of the kind $\{(op_i, target_i)\}_{i=0}^{N}$, where $op_i \in \{block, if, goto, slot, end\}$ is a type of the i^{th} element, and $target_i \in \{0, \dots, N-1\}$ is a target label of the i^{th} element (it is defined for if and $goto$ elements only). A branch structure satisfies the following constraints:

1. $op_i = end$ if and only if $i = N$;
2. if $op_i \in \{if, goto\}$, then $i < N + D$;
3. $op_i = slot$ if and only if $\exists j \in \{i - D, \dots, i - 1\}\big(op_j \in \{if, goto\}\big)$.

The following features of the branch structure construction should be emphasized:

1. an abstract call is classified as if if it is supplied with a situation whose name ends with "*if-then*", e.g.:
```
beq r1, :label
  do situation('beq-if-then', ...) end
```
2. an abstract call is classified as $goto$ if it is supplied with a situation whose name ends with "*goto*", e.g.:
```
b :label
  do situation('b-goto', ...) end
```

Execution Traces Enumeration

To describe the next step, it is worth introducing the following denotations. If $op_i = slot$, then $branch_i$ denotes the index of the if or $goto$ element that proceeds the i^{th} element. If $op_i \in \{if, goto\}$, then $branch_i = i$.

Step 2 consists in enumerating execution traces for the constructed branch structure. Given a branch structure $\{(op_i, target_i)\}_{i=0}^{N}$, an execution trace is a sequence of the kind $\{(pc_i, cond_i)\}_{i=0}^{n}$, where pc_i is an integer, and $cond_i \in \{false, true\}$ satisfies the following constraints:

1. $pc_i = 0$;

2. $pc_i = N$ if and only if $i = n$;
3. if $op_{pc_i} = goto$, then $cond_{pc_i} = true$;
4. if $op_{pc_i} = block$ or
   if $op_{pc_i} \in \{if, goto\}$ and $D > 0$ or
   if $op_{pc_i} = slot$ and $pc_i < branch_i + D$:
   a. $pc_{i+1} = pc_i + 1$;
5. if $op_{pc_i} \in \{if, goto\}$ and $D = 0$ or
   if $op_{pc_i} = slot$ and $pc_i = branch_i + D$:
   a. if $cond_{branch_i} = false$:
      i. $pc_{i+1} = pc_i + 1$;
   b. if $cond_{branch_i} = true$:
      i. $pc_{i+1} = target_{branch_i}$;

Execution traces are enumerated in a random order. To guide the trace enumeration process, the following parameters are used:

- $branch\_exec\_limit$ specifies the maximum number of occurrences of an if or $goto$ element in an execution trace:
  - a trace, where there exists an index i, such that $op_i \in \{if, goto\}$ and the number of i's occurrences is greater than $branch\_exec\_limit$, is rejected;
  - the default value is 1.
- $trace\_count\_limit$ specifies the upper bound for the number of execution traces to be returned:
  - if $trace\_count\_limit \neq -1$ and the number of execution traces (for a given value of the $branch\_exec\_limit$ parameter) exceeds $trace\_count\_limit$, the engine returns the first $trace\_count\_limit$ traces;
  - the default value is -1 (no limit).

Test Sequence Concretization

Step 3 consists in concretizing the test sequence for the given execution trace. It includes two stages:

1. construction of control code;
2. construction of initialization code.

Stage 1 consists in injecting a special code, so-called control code, into $block$ and $slot$ elements of the branch structure. The goal is to guarantee that the test sequence is executed as it is specified in the execution trace. In other words, the control code changes the registers used by the conditional branches in such a way that every time the instruction is executed, the calculated condition is equal to the one specified in the execution trace.

The control code construction is based on *data streams* specified as values of the "*stream*" parameter of the situations of the conditional branches, e.g. (MIPS):

```
beq r1, :label
  do situation('beq-if-then', :stream => 'stream1') end
```

A data stream can be thought as a pair $\langle data, i \rangle$, where $data$ is an array, and i is an index. Three code patterns expressed in the terms of the target assembly language are defined for each data stream type: $init = \{i = 0;\}$, $read(r) = \{r = data[i]; i = i + 1;\}$, and $write(r) = \{data[i] = r;\ i = i + 1;\}$. These patterns compose a *stream preparator*, e.g. (ARM):

```
stream_preparator(:data_source => 'REG', :index_source => 'REG') {
  init {
```

```
    adr index_source, start_label
  }

  read {
    ldar data_source, index_source
    add index_source, index_source, 8, 0
  }

  write {
    stlr data_source, index_source
    add index_source, index_source, 8, 0
  }
}
```

A data stream instance is declared as follows (ARM):

```
data {
  # Array storing the values of the register used by the conditional branch
  label :branch_data
  dword 0x0, 0x0, 0x0, 0x0, ...
}
...
# Stream  Label          Data  Address  Size
stream   :branch_data,   x0,   x10,     128
```

Stage 2 consists in constructing an initialization code. It also uses data streams, but in this case, the *init* and *write* patterns are applied instead of *read*.

Requirements

Currently, the *branch* test engine imposes the following requirements on test templates:

1. Each conditional branch instruction should be supplied with a situation whose name ends with "*if-then*", e.g. (MIPS):
   ```
   beq r1, :label
     do situation('beq-if-then', ...) end
   ```
2. The *stream* parameter is obligatory for situations of conditional branch instructions, e.g. (MIPS):
   ```
   beq r1, :label
     do situation('beq-if-then', :stream => 'branch_data') end
   ```
3. A conditional branch instruction's register should coincide with the *data_source* register of the corresponding data stream.
4. *data_source* and *index_source* registers of data streams should be pairwise different.
5. *index_source* registers should not be used inside test templates.
6. Each unconditional branch instruction should be supplied with a situation whose name ends with "*goto*", e.g. (MIPS):
   ```
   b :label
     do situation('b-goto', ...) end
   ```

Limitations

1. Situations of non-branch instructions are ignored.

Example

Here is an example illustrating how the *branch* test engine works (ARM).

```
class BranchGenerationTemplate < ArmV8BaseTemplate
  def pre
```

```
    super

    data {
      org   0
      align 8

      # Arrays storing the values of the registers of the conditional
branches
      label :branch_data_0
      dword 0x0, 0x0, 0x0, 0x0,    0x0, 0x0, 0x0, 0x0,
            0x0, 0x0, 0x0, 0x0,    0x0, 0x0, 0x0, 0x0,
            0x0, 0x0, 0x0, 0x0,    0x0, 0x0, 0x0, 0x0,
            0x0, 0x0, 0x0, 0x0,    0x0, 0x0, 0x0, 0x0

      label :branch_data_1
      dword 0x0, 0x0, 0x0, 0x0,    0x0, 0x0, 0x0, 0x0,
            0x0, 0x0, 0x0, 0x0,    0x0, 0x0, 0x0, 0x0,
            0x0, 0x0, 0x0, 0x0,    0x0, 0x0, 0x0, 0x0,
            0x0, 0x0, 0x0, 0x0,    0x0, 0x0, 0x0, 0x0
    }

    # Data stream preparator
    stream_preparator(:data_source => 'REG', :index_source => 'REG') {
      init {
        adr index_source, start_label
      }

      read {
        ldar data_source, index_source
        add index_source, index_source, 8, 0
      }

      write {
        stlr data_source, index_source
        add index_source, index_source, 8, 0
      }
    }
  end

  def run
    # Data stream instances for the conditional branches
    stream :branch_data_0, x0, x10, 32
    stream :branch_data_1, x1, x11, 32

    # Request to the branch test engine:
    # select one execution trace,
    # where each branch instruction is called at most twice
    sequence(:engine => 'branch',
             :branch_exec_limit => 2,
             :trace_count_limit => 1) {
      label :label0
        nop
        cbnz x0, :label0
          do situation('cbnz-if-then', :stream => 'branch_data_0') end
        nop
        cbz  x1, :label1
          do situation('cbz-if-then',  :stream => 'branch_data_1') end
```

```
        nop
        b_imm    :label0
          do situation('b-goto') end
      label :label1
        nop
    }.run 10 # Repeat 10 times
  end
end
```

In Step 1, a branch structure is constructed (for ARM, $D = 0$):

0. $(block, \phi)$ `nop`
1. $(if, 0)$ `cbnz x0, :label0`
2. $(block, \phi)$ `nop`
3. $(if, 6)$ `cbz x1, :label1`
4. $(block, \phi)$ `nop`
5. $(goto, 0)$ `b_imm :label0`
6. $(block, \phi)$ `nop`
7. (end, ϕ)

In step 2, a single execution trace ($trace\_count\_limit = 1$), such that the number of calls of each branch instruction does not exceed two ($branch\_exec\_limit = 2$), is selected, e.g., $\{0, 1, 0, 1, 2, 3, 6, 7\}$:

0. $pc_0 = 0$: $(block, \phi)$ `nop`
1. $pc_1 = 1$: $(if, 0)$ `cbnz x0, :label0`
2. $pc_2 = 0$: $(block, \phi)$ `nop`
3. $pc_3 = 1$: $(if, 0)$ `cbnz x0, :label0`
4. $pc_4 = 2$: $(block, \phi)$ `nop`
5. $pc_5 = 3$: $(if, 6)$ `cbz x1, :label1`
6. $pc_6 = 6$: $(block, \phi)$ `nop`
7. $pc_7 = 7$: (end, ϕ)

In step 3, the test sequence is concretized — a control and initialization code is constructed:

```
      // Initialization code (preparation)
      adr x10, branch_data_0    // init(branch_data_0)
      movz x0, #0x938f, LSL #0  // cbnz: true
      stlr x0, [x10, #0]        // write(branch_data_0)
      add x10, x10, #8, LSL #0
      movz x0, #0x0, LSL #0     // cbnz: false
      stlr x0, [x10, #0]        // write(branch_data_0)
      add x10, x10, #8, LSL #0
      adr x10, branch_data_0    // init(branch_data_0)
      movz x1, #0x0, LSL #0     // cbz: true

      // Test sequence (stimulus)
label0_0000:
      nop
      ldar x0, [x10, #0]        // Control code
      add x10, x10, #8, LSL #0  // read(branch_data_0)
      cbnz x0, label0_0000
      nop
      cbz x1, label1_0000
      nop
      b label0_0000
```

```
label1_0000:
      nop
```