Low-Level Program Verification using Matching Logic Reachability

Dwight Guth, Andrei Ştefănescu, and Grigore Roşu
University of Illinois at Urbana-Champaign

MODULE WCET-SYNTAX

SYNTAX *Blue* ::= *BOpCode Register*.*Exp*.*Exp* [klabel(`blue`), strict(3, 4)]

SYNTAX *BOpCode* ::= add
| sub
| mul
| div
| or
| and

SYNTAX *Exp* ::= *Register*
| # *Int*

SYNTAX *Register* ::= *r Int*

Three productions define the basic pieces of R-Type instructions in the language. The language is a 3-register language, so each instruction has two sources and one target.

The language has two indexing modes: an integer may signify the contents of a numbered register, or it may be an immediate value. We use strictness to evaluate the arguments of an instruction to their final value.

SYNTAX *Clear* ::= *ClOpCode Register*.*Exp* [klabel(`unot`, strict(2)]

SYNTAX *ClOpCode* ::= not

Here we define the first U-Type instruction: not. Instead of taking two sources and one target, it takes one source and one target.

SYNTAX *Mbop* ::= *MOpCode Exp*.*Exp* [klabel(`store`), strict(2, 3)]

SYNTAX *ClOpCode* ::= load

SYNTAX *MOpCode* ::= store

In addition to basic arithmetic and bitwise logical operations, our language supports load and store operations to index memory dynamically. Load evaluates a memory location into a register; store evaluates a memory location and a value and puts the value into memory.

SYNTAX *Jbar* ::= *JOpCode* &[klabel(`jmpz`)]

SYNTAX *JOpCode* ::= jmp

The jmp instruction takes a labelled code block and unconditionally jumps to that location.

SYNTAX *Rebne* ::= *ReOpCode Id Exp*.*Exp* [klabel(`bnez`, strict(3, 4)]

SYNTAX *ReOpCode* ::= beq
| bne
| blt
| ble

The language also supports conditional jump instructions. Each of these instructions evaluates two integers and either jumps or does not jump based on the result of an arithmetic comparison.

SYNTAX *Nbar* ::= *NOpCode*

SYNTAX *NOpCode* ::= halt

The halt instruction terminates code execution immediately, no matter what instructions follow it.

SYNTAX *Sbar* ::= *SOpCode Exp* [klabel(`sleep`, strict(2)]

SYNTAX *SOpCode* ::= sleep

The sleep instruction takes an argument specifying a length of time in cycles, and sleeps the processor until the time has elapsed.

SYNTAX *Rbar* ::= *ROpCode Register*.*Id* [klabel(`rinst`)]

SYNTAX *ROpCode* ::= rand

SYNTAX *Wbar* ::= *WOpCode Id*.*Exp* [klabel(`winst`, strict(2)]

SYNTAX *WOpCode* ::= write

SYNTAX *PWbar* ::= *PWOpCode Register*.*Id Exp* [klabel(`rwinst`, strict(4)]

SYNTAX *PWOpCode* ::= rw

These three instructions perform I/O operations on an asynchronous I/O buffer. This named buffer can be written to at any time by the environment the program is running in (simulated as a configuration parameter). The read instruction reads the value of the buffer, the write instruction writes the value of the buffer, and the rw instruction atomically first reads, then writes the value of the buffer. The buffer is initialized to 0, however a race that can occur if the program reads a value, then resets the value of the buffer, but another value is written in between.

SYNTAX *BrOpCode* ::= int

SYNTAX *NOpCode* ::= rti

Finally, we support periodic timer interrupts. The int instruction schedules an interrupt, which fires periodically until the program is terminated. The rti instruction returns from an interrupt.

SYNTAX *ClOpCode* ::= ti

SYNTAX *BrOpCode* ::= bgt
| ble

```
blt X1:Id, X2:Id
  ==> if X1 < X2 goto L
```
[macro]

```
bgt X1:Id, X2:Id
  ==> blt X2, X1
```
[macro]

```
bge X1:Id, X2:Id
  ==> blt X1, X2 ; jmp L
```
[macro]

We also define a couple convenience instructions as macros. The li instruction loads either an immediate or the contents of a register into another register. The bgt and bge instructions complete our set of conditional jump instructions.

SYNTAX *Id* ::= *main*

"main" is a special program label denoting the entry point of the program.

SYNTAX *Instr* ::= *Ibar*
| *Ibar Instr*

SYNTAX *Block* ::= *{ Instr }*

SYNTAX *Blocks* ::= *Block*
| *Block Blocks*

A program consists of a list of labelled code blocks each containing a sequence of instructions.

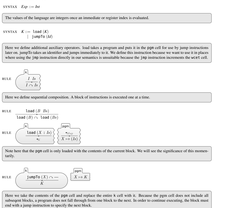
SYNTAX *Ibar* ::= *Rbar*
| *Ubar*
| *Mbar*
| *Bbar*
| *Abar*
| *Sbar*
| *Nbar*
| *Rbar*
| *Wbar*
| *PWbar*

SYNTAX *OpCode* ::= *BOpCode*
| *ClOpCode*
| *MOpCode*
| *ReOpCode*
| *JOpCode*
| *NOpCode*
| *SOpCode*
| *ROpCode*
| *WOpCode*
| *PWOpCode*

END MODULE

MODULE WCET

CONFIGURATION:



This configuration declaration specifies the initial configuration of the program by means of several variables that are initial (and by the rest) after being specified by the user on the command line. This initial configuration performs preprocessing on the program to load it into the pgm cell and then jumps to the entry point, labelled main.

SYNTAX *RBrnah* ::= *Int*

SYNTAX *Exp* ::= *Int*

The values of the language are integers once an immediate or register index is evaluated.

SYNTAX *K* ::= load *{K}*
| jumpTo *{id}*

Here we define additional auxiliary operators. load takes a program and puts it in the pgm cell for use by jump instructions later on. jumpTo takes an identifier and jumps immediately to it. We define this instruction because we want to use it in places where using the jmp instruction directly in our semantics is unsuitable because the jmp instruction increments the wcet cell.

RULE
```
X:Int
~>
.K
```
[X := .K]

Here we define sequential composition. A block of instructions is executed one at a time.

RULE
```
load (I:Int => load (Ib)
```

RULE
```
load X:Int
~>
.K
```
[pgm: .Map]
```
.K
~>
X ↦ I:Int
```

Note here that the pgm cell is only loaded with the contents of the current block. We will see the significance of this momentarily.

RULE
```
jumpTo(X:Int) ~>
.K
```
[pgm: .Map]
```
X ↦ K
```

Here we take the contents of the pgm cell and replace the entire k cell with it. Because the pgm cell does not include all subsequent blocks, a program does not fall through from one block to the next. In order to continue executing, the block must end with a jump instruction to specify the next block.

RULE
```
I:I
~>
.K
```
[reg: .Map]
```
I:I ↦ I:Int
```

Register lookup looks up the numbered register in the reg cell.

RULE
```
# I
~>
I
```

And an immediate evaluates to its own declared value.

RULE
```
add X:I, Y:I, Z:I
time (.wAdd)
```
[reg: .Map]
```
R
R[I:Int ↦ Z:I ]
```

RULE
```
sub X:I, Y:I, Z:I
time (.wSub)
```
[reg: .Map]
```
R
R[I:Int ↦ Z:I ]
```

RULE
```
mul X:I, Y:I, Z:I
time (.wMul)
```
[reg: .Map]
```
R
R[I:Int ↦ Z:I ]
```

RULE
```
div X:I, Y:I, Z:I
time (.wDiv)
```
[reg: .Map]
```
R
R[I:Int ↦ Z:I ]
```

RULE
```
or X:I, Y:I, Z:I
time (.wOr)
```
[reg: .Map]
```
R
R[I:Int ↦ Z:I ]
```

RULE
```
and X:I, Y:I, Z:I
time (.wAnd)
```
[reg: .Map]
```
R
R[I:Int ↦ Z:I ]
```

RULE
```
not I:I, Z:I
time (.wNot)
```
[reg: .Map]
```
R
R[I:Int ↦ Z:I ]
```

The first instruction of the language are defined above. Once the arguments are evaluated, the arithmetic operation is performed and the result is stored in the target register. Then the time of the instruction is made to elapse. The auxiliary operation time elapses time according to the specified length of the instruction in question.

RULE
```
load X:I, I:I
time (.wLoad)
```
[pgm: .Map]
```
R
R[I:I ]
```

The mem cell contains memory. An integer location in memory is evaluated, indexed in memory, and the value found there is put in a register.

RULE
```
store X:I, I:I
time (.wStore)
```
[mem: .Map]
```
M
M[I:I ]
```

STORE evaluates two arguments. The first expresses the location in memory to write to, and the second expresses the value to write there. Once evaluated, we update the mem cell with the value in memory.

RULE
```
jmp .I
time (.wJmp) ~> jumpTo I:I
```

The main semantics of jmp are the jumpTo operation. The instruction takes time, however.

RULE
```
beq X:I, I:I
time (.wBeq) ~> branch (I =/=Int X:I, I:I)
```

RULE
```
bne X:I, I:I
time (.wBne) ~> branch (I =/=Int X:I, I:I)
```

RULE
```
blt X:I, I:I
time (.wBlt) ~> branch (I <Int X:I, I:I)
```

RULE
```
ble X:I, I:I
time (.wBle) ~> branch (I ≤Int X:I, I:I)
```

RULE
```
K ~> branch (Bool.fd)
```

RULE
```
branch (true, I:I
        jumpTo (K:I)
~>
branch (false, _)
~>
```

Conditional jump evaluates a boolean condition. If the condition is true, we jump to the target. If it is false, we fall through to the next instruction.

RULE
```
halt ~>
time (halt)
```

halt removes the entire k cell, effectively ending execution (once it is finished executing).

RULE
```
sleep I
waitFor (Timeing(0))
```

The auxiliary operation waitFor elapses a specified length of time.

RULE
```
read .I, X
time (.wRead)
```
[reg: .Map, mem: .Map]
```
Reg
Reg[X ↦ I]
```

An I/O read looks up the current value of the data in the status cell, then writes it to a register.

RULE
```
write X:I, .
time (.wWrite) ~> X
```
[status: .Status]
```
Status
Status[X ↦ ]
```

write evaluates a value, then writes it to the status cell.

RULE
```
rw X:I, I:I
time (.rw) ~>
I:I
```
[status: .Status, reg: .Map]
```
X ↦ .
Reg
Reg[I ↦ I]
```

rw simply combines these two instructions together.

RULE
```
K ~> (.Id, Int.Id)
```

RULE
```
int X:I, I:I
time (.wInt) ~> (X, I +Int Time:Id)
```
[mem: .Map, int: .Set]
```
M
Time
```

int schedules an interrupt I cycles after executing, to execute periodically every I cycles thereafter. The timers cell stores the currently activated interrupts in a list of tuples.

RULE
```
K ~> (K, Int)
```

RULE
```
rti ~> ...
time (.rti) ~> K
```
[int: .Set, reg: .Map]
```
(K, Priority) ...
...
Priority
```

To return from an interrupt, we restore the previously executing code from the stack cell, which also contains the previously executing priority to restore to the priority cell. Interrupts are assigned memory priority in the order they are scheduled by the program, and can interrupt only code running at a lower priority. Recall the program begins executing at priority 0.

SYNTAX *K* ::= time *(OpCode)*

RULE
```
time (I)
waitFor (Timeing(I))
```
[timing: .Timing]

For modularity, we allow the time each instruction executes for to be specified dynamically in the Timing cell.

SYNTAX *K* ::= waitFor *(Int)*

RULE
```
waitFor (I)
updateStatus (I2) ~> updateTimers (I, .) ~> interrupt (I, .)
```
[int: .Set, timing: .Timing]
```
I
I2 +Int I
```

The wcet cell stores the length of time the program has already been running for, in total. When time passes, two types of things happen: updateStatus and time. The first is a write on an I/O line, provided in advance in the Inbuf cell. The second is the firing of a timer interrupt.

SYNTAX *K* ::= updateStatus *(Int Map)*

SYNTAX *K* ::= *Int.Map()*

RULE
```
updateStatus (Ibuf)
```
[int: .Set, status: .Status]
```
I
Status
Status[M]
```
requires $I \geq_{Int} Ibuf \wedge_{Bool} I \leq_{Int} Start$

RULE
```
updateStatus (Ibuf)
~>
```
[int: .Set]
```
I
```
requires $\neg_{Bool} (I \geq_{Int} Ibuf \wedge_{Bool} I \leq_{Int} Start)$

RULE
```
updateStatus (_, .)
~>
```
[status: .Status]
```
.
```

Essentially, updateStatus processes every input event between time Start and time I. Each event contains a map of associated I/O writes, which is merged with the status cell. Events occuring before or after the time window of the instruction are ignored.

SYNTAX *K* ::= updateTimers *(List)*

RULE
```
updateTimers ((X, Expires, Interval) => ...)
~>
```
[int: .Set]
```
X
```
requires $I \geq_{Int} Expires \wedge_{Bool} Interval =/=_{Int} 0$

RULE
```
updateTimers ((X, Expires, Interval) => ...)
~>
```
[int: .Set]
```
X
```
requires $I \geq_{Int} Expires \wedge_{Bool} Interval =/=_{Int} 0$

RULE
```
updateTimers (_, .)
~>
```

Essentially, updateTimers updates the interrupts cell with all the interrupts that have been triggered by the previous instruction. When an interrupt is triggered, the tuple containing info on the interrupt is updated with a new expiration time.

SYNTAX *K* ::= interrupt *(List.Int)*

RULE
```
interrupt ((X, _, _) => ...) => .)
~>
```
[int: .Set]
```
X
```
requires $\neg_{Bool} X \in_{Int} S) \wedge_{Bool} N \leq_{Int} Priority$

RULE
```
interrupt ((X, _, _) => ...) => K)
~>
```
[int: .Set, pstack: .List]
```
Priority (X)
S
```
requires $N >_{Int} Priority \wedge_{Bool} X \in_{Int} S$

RULE
```
interrupt (_, _) => ...)
~>
```
[int: .Set, pstack: .List]
```
P ::_{Int} X
(X, Priority)
```

Essentially, interrupt fires exactly one pending interrupt, removing it from the interrupts cell. This only occurs if there is a pending interrupt that is not already executing or higher than the current priority of the process, however. When an interrupt is fired, we put the current priority and the current instruction on the stack, and update the new priority. Note that the priority of an interrupt is simply the number of interrupts that were scheduled after the interrupt in question. Thus if we schedule 3 interrupts, the first has priority 3, the second has priority 4, etc. User code has priority 0. Once an interrupt reaches an rti instruction, we return to the previous priority and instruction. Note that this allows interrupts to interrupt other interrupts of lower code resume. Thus, like a real assembly language, we can starve low-priority code by firing too many high-priority interrupts.

SYNTAX *Id* ::= *Token* [*klabel*(`)`)]

END MODULE