An Esterel Virtual Machine for Embedded Systems

Becky Plummer  Mukul Khajanchi  Stephen A. Edwards

Columbia University
An Esterel Virtual Machine

Goal: Run big Esterel programs in memory-constrained settings.

Our target: the Hitachi H8-based RCX Microcontroller for Lego Mindstorms
An Example

module Example:
input I, S;
output O;
signal R, A in
  every S do
    await I;
    weak abort
    sustain R
    when immediate A;
    emit O
  loop
    pause; pause;
    present R then
      emit A
    end present
  end loop
end every
end signal
end module
Challenges

Esterel’s semantics require any implementation to deal with three issues:

- Concurrent execution of sequential threads of control within a cycle
- The scheduling constraints among these threads due to communication dependencies
- How control state is updated between cycles
How did we handle them?

- A virtual machine specifically designed to support Esterel features
- A sequentializing algorithm
- Conversion from GRC to BAL and then to a compact byte code
Phase 1: Schedule
Phase 2: Assign Threads

An Esterel Virtual Machine for Embedded Systems – p. 7/4
Phase 3: Sequentialize
Phase 4: Add Labels

Add Labels

jmp done

done

case 1

An Esterel Virtual Machine for Embedded Systems – p. 9/44
Phase 5: Convert to BAL

```
t0
  STHR 1 t1
  EMT 1
  SWC 1
  STHR 1 NR1
  END

NR1
  SWCU
t1
  TWB 2 2 case_1
  JMP done
  case_1
done
  SWC 0
```
Phase 6: Convert to Byte Code

```
t0
  STHR 1 t1
  EMT 1
  SWC 1
  STHR 1 NR1
  END

NR1
  SWCU

07 01 00 0e
04 01
05 01
07 01 00 0d
03

0c

06 00 15
05 00
```

Convert to byte code
Sequential Code Generation

1. Schedule the nodes in the graph
2. Assign thread numbers
3. Sequentialize the graph
4. Set the execution path by adding labels
5. Convert to BAL
6. Assemble to produce bytecode
Sequentialization
Sequentialization

The dotted line labeled F represents the frontier. The frontier starts at the top of the graph.
Sequentialization

The frontier moves down a node at a time in scheduled order.
When a node is in the same thread as the most recently moved one, it is simply moved above the frontier.
Sequentialization

However, when the next node is from a different thread, a switch is added to the previous thread and an active point is added to the new thread just above the just-moved node.
The algorithm is complete when the frontier has swept across all nodes in scheduled order.
Sequentializing Algorithm

1: for each thread $t$ in $G$ do
2:    create new active point $p$
3:    copy first node $n$ of $t$ in $G$ to $n'$ new node in $G'$
4:    connect $p$ and $n'$
5:    add $p$ to $P[t]$ and add $n'$ to $A[t]$
6:    $t' =$ the first thread
7: for each node $n$ in scheduled order do
8:    $t$ is thread of $n$
9:    if $t \neq t'$ then
10:       for each parent $p$ in $P[t']$ do
11:          for each successor $c$ of $p$ in $A[t']$ do
12:             create switch node $s$ from $t'$ to $t$ and connect $s$ between $p$ and $c$
13:             replace $P[t']$ with the set of new switch nodes
14:       move $n$ to $P[t]$ and remove it from $A[t]$
15:    for each unreached successor $c$ of $n$ do
16:       copy $c$ to $c'$ new node in $G'$
17:       if $n$ is a fork then
18:          add child to $A[\text{thread of } c]$
19:       else
20:          add child to $A[t]$
21:    $t' = t$ {remember the last thread}
Why VM?

- Goal: constrained-memory environment
- Instruction set has direct support for Esterel constructs like concurrency, preemption, and signals
- E.g., a context switch can be specified in just two bytes
VM Details
VM Details

- Signal status registers
- Completion code registers
- Per-thread program counters
- Inter-instant state-holding registers
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMT</td>
<td>Emit a Signal</td>
<td>04 RR</td>
</tr>
<tr>
<td>SSIG</td>
<td>Clear Signal</td>
<td>0A RR</td>
</tr>
<tr>
<td>SSTT</td>
<td>Set State</td>
<td>0B RR VV</td>
</tr>
<tr>
<td>STHR</td>
<td>Set Thread</td>
<td>07 TT HH LL</td>
</tr>
</tbody>
</table>
## VM: Control Flow Instructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>END</td>
<td>Tick End</td>
<td>03</td>
</tr>
<tr>
<td>JMP</td>
<td>Jump</td>
<td>06 HH LL</td>
</tr>
<tr>
<td>NOP</td>
<td>No Operation</td>
<td>01</td>
</tr>
</tbody>
</table>
## VM: Branch, Switch, Terminate

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>MWB</td>
<td>Multiway Branch (State)</td>
<td>2D NL RR HH2 LL2 ...</td>
</tr>
<tr>
<td></td>
<td>Multiway Branch (Comp.)</td>
<td>4D NL RR HH2 LL2 ...</td>
</tr>
<tr>
<td></td>
<td>Two Way Branch (State)</td>
<td>29 RR HH LL</td>
</tr>
<tr>
<td>TWB</td>
<td>Two Way Branch (Signal)</td>
<td>49 RR HH LL</td>
</tr>
<tr>
<td></td>
<td>Two Way Branch (Comp.)</td>
<td>69 RR HH LL</td>
</tr>
<tr>
<td>SWC</td>
<td>Switch Thread</td>
<td>05 TT</td>
</tr>
<tr>
<td>SWCU</td>
<td>Switch Unknown</td>
<td>0C</td>
</tr>
<tr>
<td>TRM</td>
<td>Set Completion Code for Join</td>
<td>08 RR VV</td>
</tr>
</tbody>
</table>
switch(opcode & 0x1F) {
    ...
    case SWC:
        // Increment the program counter
        ++pc;
        // Store the current thread as the last thread
        last_thread = current_thread;
        // Get the next thread
        current_thread = *pc;
        // Increment the program counter
        ++pc;
        // Store old pc associated with the old thread
        threads[last_thread] = pc;
        // Load the pc associated with the new thread
        pc = threads[current_thread];
        break;
    ...
}
VM: Switch Unknown

... case SWCU:
    // Make the thread stored in last_thread, the current thread
    temp = current_thread;
    current_thread = last_thread;
    last_thread = temp;
    // Store old pc
    threads[last_thread] = pc;
    // Load new pc
    pc = threads[current_thread];
    break;
...
VM in action
VM in action

<table>
<thead>
<tr>
<th>t0</th>
<th>pc = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>STHR 1 t1</td>
</tr>
<tr>
<td>04</td>
<td>EMT 1</td>
</tr>
<tr>
<td>06</td>
<td>SWC 1</td>
</tr>
<tr>
<td>08</td>
<td>STHR 1 NR1</td>
</tr>
<tr>
<td>12</td>
<td>SWC 1</td>
</tr>
<tr>
<td>14</td>
<td>END</td>
</tr>
<tr>
<td>NR1</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>SWCU</td>
</tr>
<tr>
<td>t1</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>TWB 2 1 case_1</td>
</tr>
<tr>
<td>19</td>
<td>JMP done</td>
</tr>
<tr>
<td></td>
<td>case_1 done</td>
</tr>
<tr>
<td>22</td>
<td>SWC 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Threads</th>
<th>Signals</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>States</th>
<th>Joins</th>
</tr>
</thead>
<tbody>
<tr>
<td>..</td>
<td>..</td>
</tr>
</tbody>
</table>
**VM in action**

```
t0
00:  STHR 1 t1
04:  EMT 1
06:  SWC 1
08:  STHR 1 NR1
12:  SWC 1
14:  END

NR1
15:  SWCU

   t1
16:  TWB 2 1 case_1
19:  JMP done
     case_1
done
22:  SWC 0

pc = 4
last_thread = 0
Threads  Signals
       0     0
       16    0

States  Joins
        ..    ..
```
VM in action

```
00: STHR 1 t1
04: EMT 1
06: SWC 1
08: STHR 1 NR1
12: SWC 1
14: END

NR1
15: SWCU

pc = 6
last_thread = 0

Threads	Signals
0	0
16	1

States	Joins
.. ..

pc = 6
last_thread = 0
```
VM in action

```
<table>
<thead>
<tr>
<th>t0</th>
<th>pc = 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>00: STHR 1  t1</td>
<td>last_thread = 0</td>
</tr>
<tr>
<td>04: EMT 1</td>
<td></td>
</tr>
<tr>
<td>06: SWC 1</td>
<td></td>
</tr>
<tr>
<td>08: STHR 1 NR1</td>
<td></td>
</tr>
<tr>
<td>12: SWC 1</td>
<td></td>
</tr>
<tr>
<td>14: END</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NR1</th>
<th>Threads</th>
<th>Signals</th>
</tr>
</thead>
<tbody>
<tr>
<td>15: SWCU</td>
<td>8</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>t1</th>
<th>States</th>
<th>Joins</th>
</tr>
</thead>
<tbody>
<tr>
<td>16: TWB 2 1 case_1</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>19: JMP done</td>
<td>..</td>
<td></td>
</tr>
<tr>
<td>case_1 done</td>
<td>..</td>
<td></td>
</tr>
</tbody>
</table>

| 22: SWC 0     |
```

An Esterel Virtual Machine for Embedded Systems – p. 32/4
VM in action

```

10: STHR 1 t1
04: EMT 1
06: SWC 1
08: STHR 1 NR1
12: SWC 1
14: END

NR1
15: SWCU

10: TWB 2 1 case_1
19: JMP done

pc = 19
last_thread = 0

Threads  Signals
8        0
16       1

States  Joins
.. ..

pc = 19
last_thread = 0
```
VM in action

t0
00: STHR 1 t1
04: EMT 1
06: SWC 1
08: STHR 1 NR1
12: SWC 1
14: END

NR1
15: SWCU
t1
16: TWB 2 1 case_1
19: JMP done

case_1
done
22: SWC 0

pc = 22
last_thread = 0

Threads  Signals

8  0
16  1

States  Joins

.  .
VM in action

00: STHR 1 t1
04: EMT 1
06: SWC 1
08: STHR 1 NR1
12: SWC 1
14: END

NR1
15: SWCU

pc = 8
last_thread = 1

Threads | Signals
---------|--------
8        | 0
24       | 1

States | Joins
--------|--------
..      | ..
VM in action

t0
00: STHR 1 t1
04: EMT 1
06: SWC 1
08: STHR 1 NR1
12: SWC 1
14: END
NR1
15: SWCU
t1
16: TWB 2 1 case_1
case_1
22: SWC 0

pc = 12
last_thread = 1

<table>
<thead>
<tr>
<th>Threads</th>
<th>Signals</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>States</th>
<th>Joins</th>
</tr>
</thead>
<tbody>
<tr>
<td>..</td>
<td>..</td>
</tr>
</tbody>
</table>
VM in action

\begin{align*}
t_0 & \\
00: & \text{STHR 1 t1} \\
04: & \text{EMT 1} \\
06: & \text{SWC 1} \\
08: & \text{STHR 1 NR1} \\
12: & \text{SWC 1} \\
14: & \text{END} \\

\text{NR1} & \\
15: & \text{SWCU} \\

t_1 & \\
16: & \text{TWB 2 1 case_1} \\
19: & \text{JMP done} \\
\text{case_1} & \\
\text{done} & \\
22: & \text{SWC 0} \\
\end{align*}

\begin{tabular}{|c|c|}
\hline
pc & 15 \\
last\_thread & 0 \\
\hline
\end{tabular}

\begin{tabular}{|c|c|}
\hline
Threads & Signals \\
14 & 0 \\
15 & 1 \\
\hline
\end{tabular}

\begin{tabular}{|c|c|}
\hline
States & Joins \\
.. & .. \\
\hline
\end{tabular}
VM in action

\[
\begin{array}{l}
\text{t0} \\
00: \quad \text{STHR 1 t1} \\
04: \quad \text{EMT 1} \\
06: \quad \text{SWC 1} \\
08: \quad \text{STHR 1 NR1} \\
12: \quad \text{SWC 1} \\
14: \quad \text{END} \\
\end{array}
\]

\[
\begin{array}{l}
\text{NR1} \\
15: \quad \text{SWCU} \\
\end{array}
\]

\[
\begin{array}{l}
\text{t1} \\
16: \quad \text{TWB 2 1 case_1} \\
19: \quad \text{JMP done} \\
\text{case_1} \\
\text{done} \\
22: \quad \text{SWC 0} \\
\end{array}
\]

\[
\begin{array}{c}
\text{pc} = 14 \\
\text{last_thread} = 1 \\
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{Threads} & \text{Signals} \\
\hline
14 & 0 \rule{0pt}{2.6ex} \\
15 & 1 \rule{0pt}{2.6ex} \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{States} & \text{Joins} \\
\hline
. . & . . \rule{0pt}{2.6ex} \\
\hline
\end{array}
\]
VM in action

```
t0
00:  STHR 1 t1
04:  EMT 1
06:  SWC 1
08:  STHR 1 NR1
12:  SWC 1
14:  END

NR1
15:  SWCU

t1
16:  TWB 2 1 case_1
data_1
19:  JMP done
case_1

done
22:  SWC 0
```

```
Threads   Signals

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
</tr>
</tbody>
</table>

States   Joins

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>..</td>
<td>..</td>
</tr>
</tbody>
</table>
```

pc = 15
last_thread = 0
The engineering details

- brickOS 2.6.10 on Redhat Linux
- gcc cross compiler 4.0.2. for H8300
- Download lx files to the lego RCX via USB IR tower
### Code Sizes

<table>
<thead>
<tr>
<th>Example</th>
<th>BAL</th>
<th>x86</th>
<th>60%</th>
<th>H8</th>
<th>57%</th>
</tr>
</thead>
<tbody>
<tr>
<td>dacexample</td>
<td>369</td>
<td>917</td>
<td></td>
<td>842</td>
<td>57%</td>
</tr>
<tr>
<td>abcd</td>
<td>870</td>
<td>2988</td>
<td>71%</td>
<td>2648</td>
<td>68%</td>
</tr>
<tr>
<td>greycounter</td>
<td>1289</td>
<td>3571</td>
<td>64%</td>
<td>2836</td>
<td>55%</td>
</tr>
<tr>
<td>tcint</td>
<td>5667</td>
<td>11486</td>
<td>51%</td>
<td>10074</td>
<td>51%</td>
</tr>
<tr>
<td>atds-100</td>
<td>10481</td>
<td>38165</td>
<td>73%</td>
<td>26334</td>
<td>60%</td>
</tr>
</tbody>
</table>

BAL: the size of our bytecode (in bytes)
x86: the size of optimized C code for an x86
H8: the size of optimized C code for an Hitachi H8
Percentages represent the size savings of using bytecode.
## Execution Times

<table>
<thead>
<tr>
<th>Example</th>
<th>x86 (µs)</th>
<th>BAL (µs)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>dacexample</td>
<td>0.06</td>
<td>1.1</td>
<td>18×</td>
</tr>
<tr>
<td>tcint</td>
<td>0.28</td>
<td>1.1</td>
<td>4×</td>
</tr>
<tr>
<td>atds-100</td>
<td>0.20</td>
<td>1.4</td>
<td>7×</td>
</tr>
</tbody>
</table>
Future Work

- Arithmetic Support
- Support for externally-called functions
Conclusions

- Simple Virtual Machine
- Compilation scheme statically schedules the concurrent behavior and generates straight-line code for each thread
- VM supports context-switching well
- Bytecode for our virtual machine is roughly half the size of optimized native assembly code generated from C
- Speed tradeoff not that bad! Between 4 and 7 times slower than optimized C code