Concurrent Programming of Microcontrollers: a Virtual Machine Approach

Steven Varoumas 1,2  Benoît Vaugon 3  Emmanuel Chailloux 1

1Laboratoire d’Informatique de Paris 6 (LIP6) - UPMC Sorbonne Universités
2Centre d’Étude et De Recherche en Informatique et Communications (CÉDRIC) - CNAM
3Unité d’Informatique et d’Ingénierie des Systèmes (U2IS) - ENSTA ParisTech

January 29th, 2016
Outline

1 Context
   - Microcontrollers
   - The virtual machine approach

2 Concurrent Programming
   - Studied models of concurrent programming
   - Synchronous programming

3 OCaLustre: synchronous programming with OCaml
   - Principles
   - Syntax and Semantics
   - Compilation

4 Conclusion
   - Benchmarks
   - Future Works
Context

Microcontrollers

- **Simplified computer (SoC)** (processor, memory, inputs/output peripherals)
- **Multiple interactions with the environment**
- **Used in (sometimes critical) embedded systems** (automobiles, trains, ...)

Example: the PIC 18F4620

- 8-bit architecture
- Max CPU Speed: 40 MHz
- RAM: 4 KiB / ROM: 64 KiB

⇒ Devices with limited resources *but* widely used for industry purposes (due to efficiency and cost).
Programming of $\mu$C: The Virtual Machine (VM) approach

Typically

Usually **programmed with C/Assembly languages**

- Low-level programming (little hardware abstraction)
- Few checks at compile-time (typing)
- Might be tedious and error prone (pointer arithmetics . . .)

Alternatively, we use a virtual machine approach:

**VM approach**

Port of the VM of a programming language directly implemented for such hardware (in C or assembly-language)

- Allows the running of bytecode of higher-level languages directly on $\mu$C (easier, safer programming)
- Increases the portability of code
- (Often) decreases the size of code
- Offers a level of abstraction over hardware
Language: OCaml

- **Multi paradigms** programming language (functional, imperative, object-oriented)
- High-level language with **powerful** constructions and **expressiveness** (functors, sum types, lambdas, objects, exceptions . . .)
- Improved **safety** by static typing with inference of types.
- Its Virtual Machine (the ZAM) is very **lightweight** and efficient and permits a port to devices with low resources.
OCaPIC: OCaml for PIC

- **OCaml virtual machine (ZAM)** implemented in PIC18 assembly.
- Allows the execution of (almost) all of the OCaml language on a PIC18.
- Comes with: simulators, library for external displays . . .
- **OCamlClean**: tool used for removing dead code and useless memory allocations → lighter bytecode

Benefits of this virtual machine approach:

- **Richer programming**: high-level, multi paradigms language → exceptions, pattern matching, ADT, polymorphism . . .
- **Hardware abstraction**: automatic memory management (garbage collector)
- **Improved safety**: Static typing . . .
- **Space saving**: the VM + compressed bytecode is often lighter than native code

Publication in PADL 2015 (B. Vaugon, P. Wang, E. Chailloux)
Example of an OCaPIC program:

```ocaml
open Pic
open Lcd
let disp = connect ~bus_size:Lcd.Four ~e:LATD0 ~rs:LATD2 ~rw:LATD1 ~bus:PORTB;; (* connects the LCD to the correct pins *)
disp.init ();
disp.config ();
disp.print_string "Hello world"
```
Hardware-level imperative programming is still possible:

```ocaml
open Pic;; (* Module containing write_reg, set_bit, RB0, ... *)
write_reg TRISB 0x00; (* Set the B port as an output *)
while true do
  set_bit RB0;
  Sys.sleep 500;
  clear_bit RB0;
  Sys.sleep 500;
done
```
Concurrent Programming

<table>
<thead>
<tr>
<th>Model of programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Embedded systems are equipped with multiple interfaces (buttons, sensors, communications with other systems . . .)</td>
</tr>
<tr>
<td>They must:</td>
</tr>
<tr>
<td>- Always be ready</td>
</tr>
<tr>
<td>- React quickly</td>
</tr>
<tr>
<td>- Do multiple things at once</td>
</tr>
<tr>
<td>→ <strong>Concurrent</strong> programming</td>
</tr>
</tbody>
</table>
Models of concurrent programming

### Studied models

- **Preemptive threading**
  - Implicit switching of threads. Usually automated by the OS (here: no OS).
  - Static analysis of programs is difficult

- **Cooperative threading (LWT library)**
  - Explicit switching of threads (a thread yields control to the scheduler)
  - Handling manually concurrency might be tedious and error-prone
  - High memory use

- **Functional Reactive Programming (React module)**
  - Communication between tasks by propagation of changes
  - Static analysis is difficult
  - Very high memory use

⇒ Unsuitied

### Chosen model: Synchronous Programming

- **Suitable**: Originally designed for the programming of critical embedded systems
- **Deterministic**: Easy ways of applying static analysis (causality).
Synchronous Programming

**The Synchronous Paradigm**

- Outputs are **simultaneous** with inputs (synchronous hypothesis)
- The various processes are run at the same (logical) time
  - A global clock slices the time in multiple intervals (discrete instants)

⇒ Extension of OCaml to this model

**Data-flow model of synchronous programming (cf. Lustre / Lucid Synchrone)**

We chose this model because:
- Every pin has a value at each instant (0 or 1) → binary flow
- Suitable with the representation of an electrical circuit
- Lightweight compilation model
OCaLustre: a prototype of data-flow extension for OCaml

OCaLustre

- An OCaLustre program is composed of OCaml functions and "Lustre" nodes.
- A node is a synchronous component that (instantaneously) computes output flows based on the values of input flows.
- Possible calls to OCaml functions from an OCaLustre node ("call (aux 1 2)").

```ocaml
let max x y = if x > y then x else y

let%node foo (x,y,z) (u,v) =
  u := 2 * v;
  v := if x then y else (call (max z 10))
```

NB: No need for type informations in OCaLustre (inferred at compile-time)
Flows

The **values used inside the nodes are data flows**, i.e sequences of values that can change through time.

- The $v$ variable is the sequence $(v_{t_1}, v_{t_2}, \ldots, v_{t_i}, \ldots)$
- The $2$ constant is the sequence $(2, 2, 2, 2, \ldots, 2, \ldots)$

Operators

"Traditional" arithmetics and boolean operators ($+, *, -, /, or, and, \ldots$) and two temporal operators:

- The memory operator $\text{pre}$:
  - $\triangledown$ at the instant $n$, $\text{pre} \ a = a_{t(n-1)}$

- The initialization operator $\rightarrow$:
  - $\triangledown \ a \rightarrow b = (a_{t_1}, b_{t_2}, b_{t_3}, \ldots, b_{t_i}, \ldots)$

Example: integers values

```ocaml
let%node numbers () (n) =
    n := 0 --> pre n + 1
```
OCaLustre : Lustre / Scade equivalence

**Figure : OCaLustre code**

```ocaml
let%node foo (a,b) (c,d) =
  d := 5 + c;
  c := (a / 2) + (b * 6)
```

**Figure : Lustre code**

```luster
node foo (a:int;b:int) returns (c:int;d:int);
let
  d = 5 + c;
  c = (a / 2) + (b * 6);
tel;
```

**Figure : Scade circuit**
OCaLustre: compilation

“single-loop” model of compilation (Lustre)

- Each node is compiled into an OCaml function.
- The generated “pure” OCaml code can be used with OCaPIC, or any other OCaml VM.
Adding a concurrent behaviour to an OCaLustre program is easy and straightforward:

```oca
let%node edge (x) (y) =
  y = false -> x && not pre x
```

```oca
let%node abro (a, b, r) (o) =
  o := edge (seenA && seenB);
  seenA := false -> not r && (a || pre seenA);
  seenB := false -> not r && (b || pre seenB)
```

Becomes:

```oca
let%node abcro (a, b, c, r) (o) =
  o := edge (seenA && seenB && seenC);
  seenA := false -> not r && (a || pre seenA);
  seenB := false -> not r && (a || pre seenB);
  seenC := false -> not r && (c || pre seenC)
```
Compilation

The system of equations inside each node becomes a sequence of assignments.

⇒ A scheduling pass is needed in order to put the assignments in the right order:

- Creation of a directed acyclic graph (DAG).
  → Represents the dependencies between flows.
- Inversion of the topological sorting of the graph
- Presence of loop inside the graph \( \Rightarrow \) causality loop \( \Rightarrow \) error

Original code

\[
\text{let}\%\text{node}\ foo (a,b,c) (d,e,f) = \\
\quad d := f \times 2; \\
\quad f := \text{if } a \text{ then } e \text{ else } 3; \\
\quad e := \text{pre } f + b / c
\]

Code after scheduling

\[
\text{let}\%\text{node}\ foo (a,b,c) (d,e,f) = \\
\quad e := \text{pre } f + b / c; \\
\quad f := \text{if } a \text{ then } e \text{ else } 3; \\
\quad d := f \times 2
\]

Matching graph of dependencies
(gray : input flows, ignored during sort)
Génération de code

- The nodes become functions.
- The equations become assignations.
- The arithmetics and boolean operators (+, -, not, ...) do not change.
- The pre operators become OCaml references (= registers) to an Option type.
- The $x \rightarrow y$ operator becomes if !init then x else y (given init a reference to true at the first instant).

```ocaml
let%node numbers () (n) =
  n := 0 --> pre n + 1

Becomes

let numbers () =
  let init = ref true in
  let pre_n = ref None in
  let numbers_step () =
    let n = if !init then 0 else (Option.get (!pre_n)) + 1 in
    init := false;
    pre_n := (Some n);
    n in
  numbers_step
```
Results / Benchmarks

Example 1: two parallels counters

```fancy
let%node cpt () (n) =
    n := 0 --> pre n + 1

let%node main () (a,b) =
    a := cpt ();
    b := cpt ()
```

Figure: LCD (simulator)

Memory use

<table>
<thead>
<tr>
<th>Tool</th>
<th>React</th>
<th>LWT</th>
<th>OCaLustre</th>
<th>Sequential code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of the program</td>
<td>23.8 KiB</td>
<td>11.7 KiB</td>
<td><strong>7.8 KiB</strong></td>
<td>6.8 KiB</td>
</tr>
<tr>
<td>Initial dynamic allocation</td>
<td>1668 B</td>
<td>1136 B</td>
<td><strong>272 B</strong></td>
<td>150 B</td>
</tr>
</tbody>
</table>

→ Suitable model for system with scarce resources
Example 2 : a chocolate heater

[...]  
(* Temperature in celsius is (1033-ctemp)/11.67 *)  
let%node update_prop (wtemp, ctemp) (prop) =  
  new_prop := 0 --> (min (100, max (0, pre new_prop + offset)));  
  delta := min (10, max (-10, ctemp - wtemp));  
  delta2 := if delta < 0 then -delta * delta else delta * delta;  
  offset := min (10, delta2);  
  prop := new_prop / 10  
  delta :=  
let%node main (plus, minus, ctemp) (wtemp, on, heat) =  
  state := call (buttons_state plus minus);  
  on := thermo_on (state);  
  wtemp := if on then change_wtemp (state) else 0;  
  heat := if on then heat (wtemp, ctemp) else false

<table>
<thead>
<tr>
<th>Language</th>
<th>OCaml</th>
<th>OCaLustre</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Bytecode File</td>
<td>PIC Program</td>
</tr>
<tr>
<td>Size</td>
<td>268 KiB</td>
<td>27 KiB</td>
</tr>
</tbody>
</table>
Conclusion

By proposing this extension, we offer a new layer of abstraction for the programming of microcontrollers.

Future works

- Extension of the prototype:
  - Multi-clocks (in progress)
  - Flows of more complex types (lists, arrays, objects, enum types, ...)
- Language for a description of the circuit
  - Description of the components connected to the μC.
  - Automatic generation of nodes from this description
- Development of more complex applications (in robotics, home automation, ...)
- Ongoing projects of porting OCaPIC to other microcontrollers: AVR Atmel (Arduino) and STM 32 (Nucleo).