Hardware Implementation of OCaml Using a Synchronous Functional Language

Loïc Sylvestre, Jocelyn Sérot, Emmanuel Chailloux

▶ **To cite this version:**

Loïc Sylvestre, Jocelyn Sérot, Emmanuel Chailloux. Hardware Implementation of OCaml Using a Synchronous Functional Language. PADL 2024: The 26th International Symposium on Practical Aspects of Declarative Languages, Jan 2024, Londres, United Kingdom. pp.151-168, ⟨10.1007/978-3-031-52038-9_10⟩. ⟨hal-04401618⟩

HAL Id: hal-04401618

https://hal.sorbonne-universite.fr/hal-04401618v1

Submitted on 18 Jul 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

Hardware implementation of OCaml using a synchronous functional language

Loïc Sylvestre[1], Jocelyn Sérot[2], and Emmanuel Chailloux[1]

[1] Sorbonne Université, CNRS, LIP6 75005 Paris, France
{loic.sylvestre,emmanuel.chailloux}@lip6.fr
[2] Université Clermont Auvergne, CNRS, Clermont Auvergne INP, Institut Pascal, 63000 Clermont-Ferrand, France
jocelyn.serot@uca.fr

Abstract. We present a hardware implementation of the high-level multi-paradigm language OCaml using a declarative language called Eclat. Eclat is tailored for programming reactive hardware applications mixing interaction with physical devices and long-running computations. It is compiled to synthesizable hardware descriptions for configuring Field Programmable Gate Arrays (FPGAs).
We have implemented the OCaml Virtual Machine as an Eclat function to execute complex computations (programmed in OCaml) in reactive applications (programmed in Eclat). This implementation comprises a bytecode interpreter and a runtime system with automatic memory management. The OCaml programmers can customize this runtime by defining external Eclat functions, *i.e.*, hardware accelerators.

Keywords: synchronous programming · functional programming · language design and implementation · FPGA · virtual machine · OCaml

1 Introduction

When programming hardware applications on FPGAs, a classical issue is to reconcile *reactivity* and *expressiveness*.

Reactivity is the ability of an application to respond "quickly enough" to any stimulus occurring from its environment. By contrast, expressiveness refers to the use of abstraction barriers (e.g., high-level programming features with automatic memory management) to hide many low-level implementation details. In particular, the timing behavior of the applications is often left unspecified.

Ensuring reactivity is a founding principle of the so-called synchronous languages [21] (e.g., Lustre [7], Esterel [4] and Signal [13]), which are based on a logical notion of time known as the *Synchronous hypothesis*. In these languages, program execution is divided into a discrete sequence of computation steps separated by clock ticks. Each computation step is logically instantaneous: it processes current inputs and produces outputs before acquiring new inputs at the next clock tick. This offers a proven methodology for designing reactive embedded applications [9].

Synchronous programs can be translated into synchronous hardware, implemented on FPGAs, by associating the *clock tick* of the synchronous model to the *global clock* of the FPGA circuit [3][22]. However, any computation having a long response time must be reformulated (by the programmer) as a sequence of instantaneous computation steps: this is cumbersome and error-prone.

The goal of this paper is to help declarative programmers write reactive applications on FPGAs. We base our work on a new approach [25] for mixing synchronous interaction (modeled as instantaneous functions) and long-running computations (as non-instantaneous functions, *e.g.*, tail-recursive functions); the whole being compiled to VHDL hardware descriptions.

We add to this language more programming features; and implement, on top of it, an OCaml virtual machine (VM) with automatic memory management. This really allows the programmer to mix synchronous interaction on the one hand and complex computations on the other hand; these computations being expressed in OCaml to be compiled by the OCaml bytecode compiler, stored in on-chip memory and executed by the VM circuit.

The contributions of the paper are:

1. the design and implementation of the Eclat language, which extends the work described in [25] with: coarse-grained parallelism, global arrays, sized integers, and a full support for FPGA synthesis; Eclat includes a dedicated construct **exec** that bridges interaction and long-running computation;
2. implementing the OCaml VM as an Eclat function (*i.e.*, a hardware accelerator) using the **exec** construct; automatic memory management is realized by a garbage collector algorithm programmed in Eclat; The OCaml stack and heap are implemented in on-chip memory as an Eclat global array;
3. implementing communication and acceleration, both enabling:
   (a) the programmer to extend the OCaml runtime with Eclat external functions using the OCaml Foreign Function Interface (FFI);
   (b) reactive applications, written in Eclat, to call the bytecode interpreter.

The resulting framework paves the way for future developments of reactive applications on FPGAs using the well-established language OCaml.

The source code of this work is available online:

https://github.com/lsylvestre/PADL24

The paper gives a general view of all the programming possibilities of the proposed approach: from the traditional hardware description, to the execution of OCaml bytecode by a VM circuitry involving custom hardware accelerators, within a reactive FPGA application.

We first present the Eclat language (section 2); then we describe our implementation of the OCaml VM in Eclat (section 3) and evaluate it in terms of efficiency and expressiveness (section 4). We show the interoperability possibilities of this VM implementation (section 5). We discuss related work (section 6) and we finally highlight future research perspectives (section 7).

2 Hardware design using Eclat

Eclat is a synchronous functional language for programming FPGAs. It can be seen both as a hardware description language (with a fine-grained control on timing and parallelism in the applications) and a general-purpose programming language for designing hardware accelerators.

2.1 Circuits as functions

Eclat is based on a call-by-value λ-calculus with let-bindings, booleans, conditional, integers and tuples. It is statically typed with let-polymorphism. Eclat functions are unary. They can be applied to tuples of values and can have functional parameters.

The execution of Eclat programs is driven by the global clock of the FPGA target. Each Eclat program is a sequence of top-level definitions (functions and global data) including a function `main`. This function `main` is implicitly mapped to physical devices (sensors or actuators) and called at each clock tick. It is said *instantaneous* because, each time it is called with input values from its environment, it returns output values before the next clock tick.

Combinational circuits can easily be described as Eclat instantaneous functions. For instance, Fig. 1b defines a classical combinational circuit `full_add` composing two instances of a sub-circuit `half_add` (Fig. 1a).

Circuit `full_add` (Fig. 1b) sums two 1-bit integers given a carry input `ci`; it returns a result `s` and a carry output `co`. The two instances of `half_add` are duplicated at compile time (by function inlining) to form the resulting hardware description. Function `main` can be mapped to platform-specific I/Os blocks, *e.g.*, three buttons and two LEDs to visualize the behavior of circuit `full_add`.

Types signatures are inferred by the Eclat compiler. For example, function `half_add` (Fig. 1a) has type `(bool * bool)` $\Rightarrow$ `(bool * bool)`. Type constructor $\tau \Rightarrow \tau'$ denotes an instantaneous function.

```
val half_add : (bool * bool) ⇒ (bool * bool)
val full_add : (bool * bool * bool) ⇒ (bool * bool)
val main : (bool * bool * bool) ⇒ (bool * bool)
```

```
let half_add(a,b) =
  let s = a xor b in
  let co = a & b in
  (s,co)
```

(a)

```
let full_add(a,b,ci) =
  let (s1,c1) = half_add(a,b) in
  let (s,c2) = half_add(ci,s1) in
  let co = c1 or c2 in
  (s, co)

let main(buttons) = full_add(buttons)
```

(b)

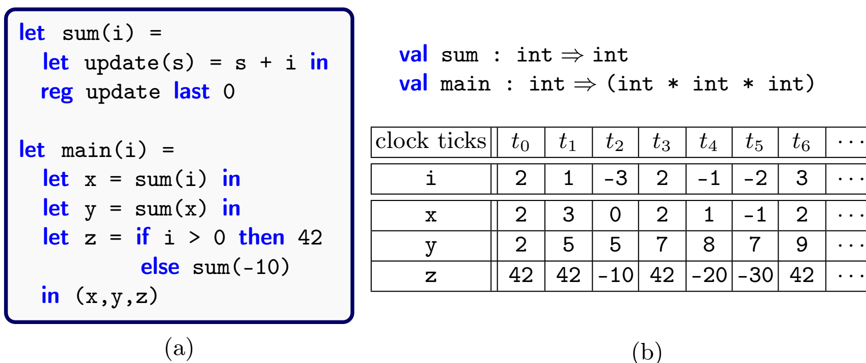
Fig. 1: A combinational circuit in Eclat

2.2 Sequential circuits

In order to design sequential circuits, ECLAT is enriched with a programming construct **reg**. This construct enables values to be memorized between successive calls to the function `main` of a given ECLAT program: (**reg** f **last** e_0) represents a register (*i.e.*, a local state) initialized with the instantaneous expression e_0 and updated with instantaneous function f; (**reg** f **last** e_0) is an expression which returns a value: the value of the associated register after updating it with f.

Each function call behaves like circuit instantiation, duplicating (at compile time) the registers that are defined in the body of the callee function. This means that the registers are *not shared* among instances of a same function; they are local and initialized only once.

Fig. 2a defines for example a function `sum` computing a cumulative sum of the values on its input `i`. A function `main` is also defined to illustrate the behavior of function calls. The chronogram on Fig. 2b traces a sequence of executions of function `main`. Each call to function `sum` produces an output that depends on the current input value and a local state (the register) manipulated by construct **reg** in the body of each instance of function `sum`. All these registers are initialized to 0 and updated according to the control flow of function `main` which is re-executed at each clock tick.



```
let sum(i) =
  let update(s) = s + i in
  reg update last 0

let main(i) =
  let x = sum(i) in
  let y = sum(x) in
  let z = if i > 0 then 42
          else sum(-10)
  in (x,y,z)
```

(a)

```
val sum : int ⇒ int
val main : int ⇒ (int * int * int)
```

clock ticks	t_0	t_1	t_2	t_3	t_4	t_5	t_6	···
i	2	1	-3	2	-1	-2	3	···
x	2	3	0	2	1	-1	2	···
y	2	5	5	7	8	7	9	···
z	42	42	-10	42	-20	-30	42	···

(b)

Fig. 2: A stateful function in ECLAT

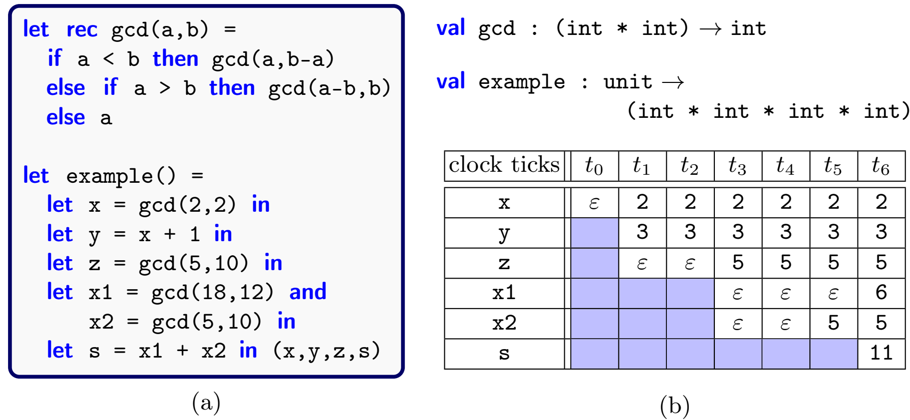
In the body of function `main`, the expression `sum(i)` updates its internal register by adding the value of its argument `i`; the value of the register is then returned and bound to name `x`. Then, the expression `sum(x)` behaves similarly: it returns a value which is bound to name `y`. Notice that each instance of `sum` refers to a different register. Conditional in ECLAT activates only one branch a time (the *then* part or the *else* part) according to the condition. Therefore, the expression (`if i > 0 then 42 else sum(-10)`) executes `sum(-10)` only when `i` is less or equal than `0`. This is similar to the *when* operator of LUSTRE [7]; however, ECLAT has call-by-value semantics; ECLAT values are not streams.

2.3 Expressing computations

Eclat provides the ML construct **let rec** for defining tail-recursive functions. Tail-recursion in Eclat can be unbounded: a pause of one clock cycle is performed at each tail-call, delaying the result of the computation.

Eclat offers also coarse-grained parallelism using a generalized form of let-bindings (**let** $x = e_1$ **and** $x_2 = e_2$ **in** e) for running the expressions e_1 and e_2 in parallel with a synchronization point (the keyword "**in**") and a continuation e.

Fig. 3a defines for instance a tail-recursive function `gcd` computing the greatest common divisor of two integer values. The figure defines also a function `example` illustrating both sequential and parallel compositions in Eclat; the chronogram on Fig. 3b traces the execution of this function.



```
let rec gcd(a,b) =
  if a < b then gcd(a,b-a)
  else if a > b then gcd(a-b,b)
  else a

let example() =
  let x = gcd(2,2) in
  let y = x + 1 in
  let z = gcd(5,10) in
  let x1 = gcd(18,12) and
      x2 = gcd(5,10) in
  let s = x1 + x2 in (x,y,z,s)
```

(a)

```
val gcd : (int * int) → int

val example : unit →
                 (int * int * int * int)
```

clock ticks	t_0	t_1	t_2	t_3	t_4	t_5	t_6
x	ε	2	2	2	2	2	2
y		3	3	3	3	3	3
z		ε	ε	5	5	5	5
x1				ε	ε	ε	6
x2				ε	ε	5	5
s							11

(b)

Fig. 3: A long-running computation in Eclat

Symbol ε on Fig. 3b represents the value of an expression that is still running. Each empty cell represents a computation that has not yet started. First, `gcd(2,2)` is computed. It responds in one clock cycle because `gcd` is a recursive function: the direct call takes one clock cycle; the two arguments are equal and thus the body of `gcd` instantaneously returns the value 2. Once `x` is bound to value the 2, `y` is instantaneously bound to the value of `x + 1` and the computation `gcd(5,10)` starts. It takes two clock ticks (one for the direct call plus one for the recursive call). Then, two calls to `gcd` are performed in parallel[3]: `gcd(18,12)` takes three clock ticks; `gcd(5,10)` takes two clock ticks and waits at the synchronization point. Then `s` is bound to `x1 + x2` and a tuple is returned.

Eclat assigns a type $\tau \to \tau'$ to non-instantaneous functions (such as `gcd`). The static type system distinguishes instantaneous functions (of type $\tau \Rightarrow \tau'$) and non-instantaneous functions (of type $\tau \to \tau'$) as explained in Sec. 2.5.

[3] At the circuit level, there are two parallel instances of `gcd`.

2.4 Mixing interaction and computation

Any expression calling a recursive function (such as `gcd` on Fig. 3a) is non-instantaneous: it cannot directly be used for real-time interaction since the inputs may be ignored and the outputs delayed while the computation is running.

This motivates the introduction of a new programming construct **exec** for executing a long-running computation step by step: (**exec** e **default** e_0) computes the expression e by performing one computation step per clock tick, with respect to the control flow, and always returns a value: either the result of the computation of the expression e if available or the value of the instantaneous expression e_0 (which provides a default value). This value is paired with a boolean *rdy* indicating when the computation of e is complete. Every time the computation of e terminates, it restarts at the next clock tick and acquires new inputs.

Fig. 4a defines, for instance, a function `main` (*i.e.*, the entry point of the Eclat program) calling both function `sum` (Fig. 2a) and function `gcd` (Fig. 3a). A chronogram is given Fig. 4b for a sequence of input values. Symbol "-" denotes the input values that are ignored by the program.

```
val main : (int * int * int) ⇒ (bool * int)
```

```
let main (i,a,b) =
  let s = sum(i) in
  let (x,rdy) = exec
                  gcd(a,b)
                default 0
  in
  let r = if rdy then x else s in
  (rdy,r)
```

(a)

clock ticks	t_0	t_1	t_2	t_3	t_4	t_5	t_6	⋯
i	2	1	-3	2	-1	-2	3	⋯
a	18	-	-	-	2	-	5	⋯
b	12	-	-	-	2	-	10	⋯
s	2	3	0	2	1	-1	2	⋯
x	0	0	0	6	0	15	0	⋯
rdy	$\bot$	$\bot$	$\bot$	$\top$	$\bot$	$\top$	$\bot$	⋯
r	2	3	0	6	1	15	2	⋯

(b)

Fig. 4: Reactive program mixing interaction and computation

As specified in the type signature, function `main` is instantaneous: it is called at each clock tick, acquiring inputs and returning outputs in a synchronous way. At each clock tick, the register associated with the instance of function `sum` is updated with input `i` and the updated value is bound to name `s`. The long-running computation `gcd(a,b)` is guarded by **exec**: it progresses step by step, *i.e.*, one step per clock tick. At time t_3 and t_5, the computation `gcd(a,b)` returns a value, which is bound to name `x`, and `rdy` is set to value $\top$ for one clock cycle. The output `r` of function `main` is defined as the value of `x` if `rdy` is $\top$, otherwise `s`.

2.5 A type system for ensuring reactivity

The static type system infers the response time of each Eclat expression using a one bit abstraction: *instantaneous* (**0**) vs. *non-instantaneous* (**1**). The typing

judgement for expression $\Gamma \vdash e : \tau|\delta$ means that, in a typing environment Γ, expression e has type τ and response time δ. Here are for instance two typing rules belonging to this type system:

$$\frac{\Gamma \vdash e_1 : \texttt{bool}|\delta_1 \qquad \Gamma \vdash e_i : \tau|\delta_i \qquad i \in \{2,3\}}{\Gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \tau|\delta_1 + \max(\delta_2, \delta_3)} \qquad \frac{\Gamma \vdash e : \tau|\mathbf{0} \qquad \Gamma(f) : \tau \Rightarrow \tau}{\Gamma \vdash \textbf{reg } f \textbf{ last } e : \tau|\mathbf{0}}$$

The left one indicates that the response time of a conditional is the response time of the condition plus the maximum response time between the *then* and *else* parts. The right one assigns an instantaneous response time to the **reg** construct but also enforces both the update function f and the initialization expression e to be instantaneous. The typing judgment for programs $\vdash \pi : \tau|\mathbf{0}$ means that program π is well-typed if it has type τ and response time $\mathbf{0}$. In other words, *well-typed programs are reactive!* Other programs are rejected at compile time.

2.6 Compilation to synthesizable hardware descriptions

Eclat is compiled to synchronous circuits by inlining all non-recursive functions and translating Eclat immediate values into statically allocated bit vectors. Tuples are also immediate values, implemented as concatenations of bit vectors.

The Eclat compiler is built as a pipeline of semantic-preserving transformations. Each input program is put in ANF-form (Administrative Normal Form)[15] (to make all the evaluation contexts explicit using let bindings), then specialized (to obtain an equivalent first-order program), and λ-lifted [14] (to make all lexical environments explicit using additional parameters, and globalize all functions). Then, non-recursive functions are inlined, and the program is translated into a Finite State Machine (FSM), each tail-recursive function definition becoming a state in this FSM.

Fig. 5 for instance gives the FSM obtained by compiling Eclat expression `(gcd(10,11) * 2)`. This FSM is depicted as a state-transition diagram. The arrows of the diagram are transitions labelled with guarded actions of the form `tick` · *`condition`*/*`action`*, meaning that *`action`* is executed at the end of the current clock tick if *`condition`* is true.

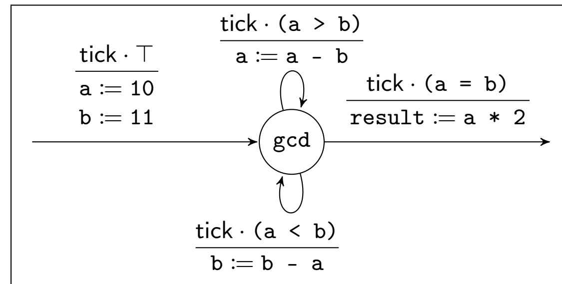


Fig. 5: FSM generated from an Eclat expression

On Fig. 5, both direct and tail-recursive calls to `gcd` are implemented as transitions (pausing for one clock cycle) with variable assignments for argument passing. The subexpression `gcd(10,11)` is first computed, then the result is instantaneously multiplied by two and assigned to variable `result`, which encodes the return value of the computation.

2.7 Expressing complex computations in Eclat

Eclat features global mutable arrays, that are implemented using on-chip memory (*i.e.*, RAM blocks available on the FPGA target). This will allow the definition of large heap and stack for the OCaml VM in Section 3. Arrays are defined using top-level declarations of the form **let static** x = c^n, where n denotes the size of the array x to be defined and c is an initial value for all its elements. Each array read x[v] takes two clock ticks (a clock tick for setting the read pointer of the RAM block and another for waiting the result provided by the interface of the RAM block). Each array modification x[v] $\leftarrow v'$ takes one clock tick (for setting both the write pointer of the RAM block and the value to be written). Parallel accesses to a given array x are forbidden and rejected at compile time[4].

We also provide assertions and print primitives for simulation purpose[5]. This is an important aspect for developing large hardware applications.

Finally, Eclat has sized integers for limiting the area of the generated circuits and implementing specific computations on large integers. Eclat integers are signed. The type constructor for integers is `int`<θ> where θ is a size variable that can be quantified by *let*. A primitive `resize_int`<n> is provided for resizing integers; it has the type scheme $\forall\theta \cdot$ `int`<θ> $\Rightarrow$ `int`<n>. Type `int` is a shortcut for `int<32>`. Typing for arithmetic operations enforces both the arguments and the result to have the same size, the result being implicitly resized. For instance:

$$\textbf{val}\ \texttt{(+)} : \forall\theta \cdot (\texttt{int<}\theta\texttt{>} * \texttt{int<}\theta\texttt{>}) \Rightarrow \texttt{int<}\theta\texttt{>}$$

Eclat does not directly support high-level features, such as general recursion and dynamic data structures with automatic memory management – features that are important in practice to implement algorithms. However, as we will see in the next sections, Eclat is sufficiently efficient for implementing a high-level language and thus enabling the programmer to express complex algorithms in it.

3 Hardware implementation of the OCaml VM in Eclat

This section presents our compilation flow for OCaml on FPGA, which consists of implementing, in Eclat, both the OCaml virtual machine and a runtime system for OCaml. We present our implementation choices for representing OCaml values and managing dynamic memory with garbage collection.

[4] This includes parallel reads due to an implementation limitation.

[5] These constructs are not synthesizable; they are all removed by the Eclat compiler, using two specific flags, when targeting FPGA synthesis tools.

3.1 The OCaml virtual machine

The OCaml VM [17] is a stack machine adapted from the Krivine machine [16] with strict application rather than call-by-name. It enables closure allocation, with an optimized execution model for ML-style unary functions to prevent the allocation of intermediate closures. The VM instruction set comprises 148 instructions manipulating a stack, a heap, a global data segment, and specialized registers such as the stack pointer.

The hardware implementation we propose closely follows the OCaml VM specification. As a limitation, it currently does not support floating point operations and object-oriented features.

3.2 Our compilation flow for OCaml on FPGA

Fig. 6 schematizes our compilation flow for OCaml on FPGA.

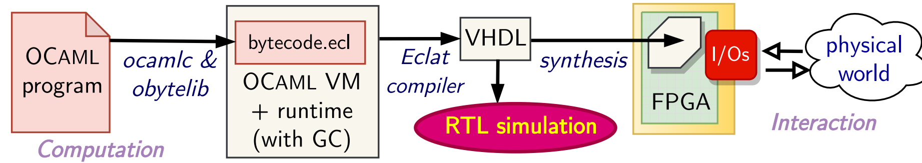


Fig. 6: Implementing OCaml on FPGA using Eclat

The bytecode produced by the OCaml bytecode compiler `ocamlc`, is first processed using the `obytelib`[6] library to obtain a more compact representation that we encode as an Eclat file named `bytecode.ecl`. This file contains:

1. an initialization function `init_data` for loading into the VM memory the global data from the OCaml program;
2. a function `external_call` for dispatching external function calls from the OCaml interpreter;
3. a global array `code` encoding each bytecode instruction of the OCaml program as a sequence of integers (*e.g.*, the sequence "19; 2" stands for instruction `POP(2)`, where 19 is the opcode of the bytecode instruction `POP`, which has one parameter: here the integer literal 2).

This generated file is added to the source code of the OCaml VM and runtime (written in Eclat). The whole application is compiled into a hardware description language (VHDL in our case) by the Eclat compiler. The resulting hardware description can be used for simulating the bytecode execution at the Register Transfer Level (RTL), *i.e.,* the computational model on which hardware description languages rely. Finally, it can be synthesized using a synthesis tool (in the work described here, we have used the Intel Quartus FPGA tool chain).

[6] https://github.com/bvaugon/obytelib

3.3 A parametrizable runtime system

The OCaml compiler adopts a uniform representation of values. Any OCaml value is either an integer (*i.e.*, an immediate value) or a pointer to a memory block. A *mark bit* is used to differentiate integers from pointers. On CPU architectures, addresses are aligned to the size of the word (e.g., 32 or 64 bits). Their least significant bit is always equal to zero. The standard implementation of the OCaml VM (called `ocamlrun`) uses this bit as the mark bit **0**. Conversely, integers have the mark bit **1**. Therefore OCaml native integers are 31-bit integers (or 63-bit integers depending on the word size) shifted to the left.

Our Eclat implementation uses a different encoding, defined Fig. 7a. Each OCaml value, represented with type `value` in Eclat, is a pair formed of one integer (of type `long`) and one mark bit (of type `bool`). The type `long` is an integer having a customizable size.

All the memory needed by the VM is statically allocated in a global array `ram` of customizable size, in which are placed global data, stack and heap (Fig. 7b). Accesses and modifications of this Eclat array are sequentialized. Each memory block allocated in this array is made of one header (encoding a size, a tag, and, possibly, additional information for garbage collection) followed by contiguous OCaml values. For example, we represent OCaml lists as either the integer 0 (representing the empty list) or a pointer to a block of three contiguous memory cells: one for the header, one for the list head and one for the list tail.

```
type long = int<31> ;;
type value = long * bool ;;

let is_int(_,b) = b ;;
let long_val(n,_) = n ;;
let val_int(n) = (n,true) ;;
```

(a)

```
let static ram = (0,true)^16384 ;;

let data_start = 0 ;;
let stack_start = 1000 ;;
let heap_start = 4000 ;;
let heap_size = 6000 ;;
```

(b)

Fig. 7: Parametrizable word size and memory partitioning for OCaml in Eclat

We have implemented a Stop&Copy garbage collector (GC) [8] for managing dynamic memory in the array `ram`. This GC requires doubling the size of the heap to organize it in two *semi-spaces*: the bytecode interpreter allocates memory blocks in one semi-space. Once the semi-space is full, the program execution is stopped in order to copy, from the current semi-space to the other, all blocks being reachable from the roots of the garbage collector (*i.e.*, the values contained in the stack, the global data segment and the registers); addresses are then updated within the remaining values; and the two semi-spaces are swapped.

4 Experimental evaluation

In this section, we evaluate the performances of our VM implementation, written in Eclat, compiled to VHDL and synthesized on an Intel Max10 FPGA. This evaluation is carried out by comparing these performances with that obtained with O2B (*OCaml on Board*)[24], an implementation of the OCaml VM written in C and running on a softcore processor also synthetized on a Max10 FPGA.

The Max10 FPGA has 50K logic cells, 200 Kbytes of on-chip memory, and a clock frequency of 50 MHz. It is embedded on a Terasic DE10-Lite FPGA board. We program it using the Intel Quartus II tool chain (version 22.1). The C code of the OCaml VM in O2B targets the Intel Nios II softcore processor and is compiled with `gcc -Os`.

Both VM implementations are configured to have 32-bit words, a stack of 3,000 words and a Stop&Copy garbage collector (GC) manipulating two semi-spaces of 6,000 words each. They use `obytelib` for preparing the OCaml bytecode generated by the OCaml Compiler (version 4.14.1).

The proposed benchmarks[7] involve the following OCaml features: tail-recursive function (Gcd), general recursive function Takeuchi (Tak), intensive composition of higher-order functions (Apply), problem solving (Queens), dynamic creation and worst-case search in a binary search tree (BST), dynamic list creation and filtering preserving sharing for a maximal sublist, using an exception (Share). Programs Tak and Queens make intensive use of the GC.

For each of these programs, we measure the speedups and resource usage (number of used logic cells and on-chip memory blocks) for our VM implementation (hereafter, VM_{Eclat}) versus that obtained with O2B.

4.1 Results

Fig. 8 reports speedups for VM_{Eclat} vs. O2B. The average speedup for the proposed benchmark is 50. This clearly shows the benefits of implementing the OCaml runtime and bytecode interpreter as a custom hardware accelerator instead of running them on a softcore processor.

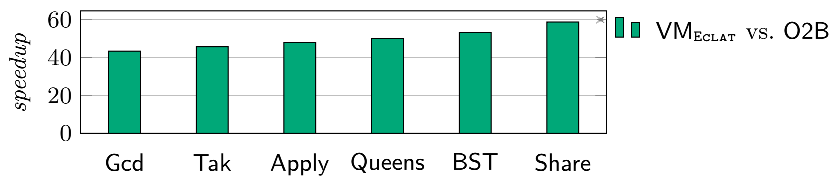


Fig. 8: Speedups achieved by our VM implementation (on the FPGA)

Our OCaml implementation (that does not embed any softcore) uses 44% of the logic cells and 33% of the on-chip memory blocks available on the Max10

[7] https://github.com/lsylvestre/PADL24/tree/main/benchs

FPGA. The softcore-based O2B implementation uses 8% of the logic cells and 66% of the on-chip memory. It is likely that the speedups related above are due, at least in part, to a better usage of the resource in logic cells by the Eclat written in Eclat.

4.2 Comparison with a PC

Being based upon OMicroB (*OCaml on Microcontroller Boards*) [27], which is a generic OCaml VM implementation for programming micro-controllers, O2B can also target a personal computer (PC). We have therefore compared the performances of our VM implementation ($\mathsf{VM}_{\text{Eclat}}$) to that obtained by OMicroB running on PC ($\mathsf{OMicroB}_{\text{pc}}$), with the same VM configuration (*e.g.*, heap size of 6,000 words). The PC used for this experiment has an Intel Core i7 with a frequency of 2.2 GHz and 16 GB of RAM.

Fig. 9 reports speedups for $\mathsf{OMicroB}_{\text{pc}}$ vs. $\mathsf{VM}_{\text{Eclat}}$. Our VM implementation $\mathsf{VM}_{\text{Eclat}}$ is around 38 times slower than $\mathsf{OMicroB}_{\text{pc}}$. This can be explained by the huge difference in frequency between the low-end FPGA used and the PC: 2.2 GHz / 50 MHz = 44. Furthermore, with these settings, OMicroB on PC is approximately 3 times slower than `ocamlrun` (the standard implementation of the OCaml VM), notably because OMicroB is space-optimized to fit on micro-controllers with few resources.

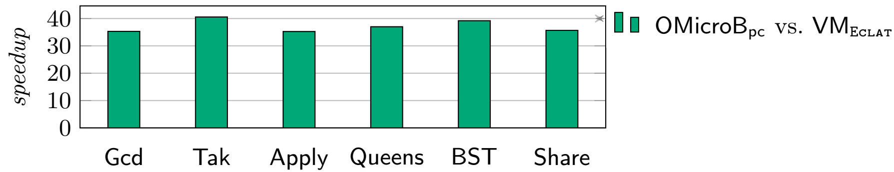


Fig. 9: Comparison between PC and FPGA execution

This section has shown the efficiency of our VM implementation targeting a small FPGA, versus another VM implementation configured in the same way and targeting a softcore processor synthesized on a same FPGA. The benchmarks presented are limited to sequential bytecode execution: in this practical case, the use of a dedicated VM circuitry cannot compensate the frequency gap between a such FPGA and a PC. The next section will show how to achieve better performances by calling external Eclat functions from OCaml bytecode.

5 Interoperability between Eclat and OCaml

For accelerating certain computations or extending the OCaml runtime with new primitives that are not definable in OCaml, we have implemented the

OCaml Foreign Function Interface (FFI). This allows OCaml programs, executed by the VM, to call external functions defined in Eclat. Furthermore, the VM itself is an Eclat function, that can be embedded in Eclat applications.

5.1 Calling Eclat functions from OCaml programs

Let's suppose that, in order to improve performances, we want to use the Eclat version of the `gcd` function defined Fig. 3 in an OCaml program instead of its direct OCaml version[8]. Function `gcd` cannot be directly applied to OCaml values since OCaml and Eclat do not have the same value representation (see the type "`value`" and primitives `long_val/val_long` defined Fig. 7a). Therefore, we use an Eclat wrapper function `gcd_glue` converting two OCaml values `v1` and `v2` (and up to five values in nested pairs) and passing them to `gcd`:

```
let gcd_glue ((v1,(v2,_)),st) =
  (val_long(gcd(long_val(v1),long_val(v2))),st)
```

By convention, such functions to be called from OCaml take also an additional parameter `st` (the state of the machine) which is also returned. Function `gcd_glue` can then be called in OCaml programs using the OCaml FFI[9], *e.g.*:

```
external gcd_ext : int -> int -> int = "gcd_glue"

let _ = List.filter (fun x -> x != 1) @@
          List.map2 gcd_ext [18;2;5;1; ··· ] [12;2;10;8; ··· ]
```

This enables significant performance gains by calling, in OCaml, hardware accelerators written in Eclat.

For instance, function `gcd_ext` is experimentally <u>65 times faster</u> than an OCaml equivalent function of type `(int -> int -> int)` compiled to bytecode and executed by our OCaml VM interpreter on the same FPGA.

Moreover, the Eclat external functions can have parallel implementations (using the Eclat **let/and/in** construct presented in Sec. 2.3) to increase speedups even further, depending on the degree of available parallelism. For instance, calling 16 times `gcd` in parallel with the same arguments is <u>1,000 times faster</u> than a pure OCaml execution using our VM implementation.

5.2 Accessing OCaml data structures from Eclat functions

Eclat external functions can operate on OCaml data structures using primitives defined in our OCaml runtime library, which also is written in Eclat.

Fig. 10 for instance defines an Eclat function `length` computing the length of an OCaml list (*i.e.*, a pointer of the type `value` defined Fig. 7a). It uses two primitives: `is_empty` (of type `value` ⇒ `bool`) to check if a list is empty, and `list_tail` (of type `value` → `value`) accessing to the tail of an OCaml list.

[8] Which, incidentally, is perfectly similar to the Eclat version.

[9] This general mechanism for making OCaml interoperable with its implementation language is very similar to that used in C implementations of OCaml on CPUs, with C external functions.

```
val length : value → int
```

```
let length(lst) =
  let rec aux(lst,acc) =
    if is_empty(lst) then acc
    else aux(list_tail(lst),acc + 1)
  in aux(lst,0)
```

Fig. 10: Eclat function computing the length of an OCaml list

Experimentally, this Eclat function `length` is 17 times faster than an equivalent OCaml version compiled to bytecode and executed by our VM implementation. This speedup is reasonable, since function `length` performs little computation (only one increment, `acc + 1`): computation time is dominated by sequential memory accesses, which take equal time in Eclat and OCaml.

5.3 Embedding OCaml code in reactive applications

Our OCaml VM is written in Eclat as an instantaneous function `ocaml_vm` (using the **exec** and **reg** constructs). Processing one instruction always takes several clock ticks. For this reason, the function `ocaml_vm` has a special output *busy* corresponding to the *rdy* output of the **exec** construct used internally. It also has an output *stop* indicating when the execution of the OCaml program is complete. Other inputs and outputs can be added to `ocaml_vm` depending on the application to be implemented.

Fig. 11 is a simple example of reactive application, written in Eclat. This application takes two buttons (`button1` and `button2`) as inputs and controls three LEDs as outputs. The entry point (function `main`) is called at each clock tick: it executes the OCaml VM (specialized for a given OCaml program) by calling function `ocaml_vm` with `button2` as an argument.

```
val ocaml_vm : bool ⇒ (bool * bool * value)
val main : (bool * bool) ⇒ (bool * bool * bool)
```

```
let main(button1,button2) =
  if button1 then (false,false,false)
  else (let (stop,busy,result) = ocaml_vm (button2) in
        let (red,green) = (busy,stop) in
        let blue = stop && (long_val(result) == 42) in
        (red,green,blue))
```

Fig. 11: A reactive Eclat application calling the OCaml VM

When `button1` is pressed, the VM execution is suspended, and the three LEDs are switched off. Otherwise, the red LED indicates if the VM is busy; the green LED indicates if the execution of the OCaml bytecode is finished; the blue LED indicates if the OCaml bytecode has produced, in a dedicated register `result`, a value equal to 42.

6 Related work

Numerous languages for hardware design have been embedded as libraries in programming environments [12]. We can cite CλASH [1] in Haskell; Hardcaml[10] in OCaml; HML in Standard ML [18]; Koika [6] or Π-ware [20] on top of the proof assistants Coq and Agda for building correct hardware. These languages exploit the expressiveness of the host programming environment, *e.g.* Coq, focussing on the effective design of hardware components, whereas our work aims at implementing high-level programming features into hardware accelerators.

Closer to our work is SAFL (*Statically Allocated Functional Language*), which is an ML-like language compiled into hardware [19]. SAFL limits the expressiveness (*e.g.*, avoiding general recursion) to offer good performances.

Shard (*Scheme on Hardware*) [23] is a hardware compiler for a functional subset of Scheme. It features parallel let-bindings and global arrays, like in our approach. It also supports closure allocation: each closure can be called only once and is immediately freed.

FHW (*Functional hardware*) [26] is a compilation flow for the intermediate representation of the Haskell Compiler GHC. It applies semantic-preserving transformations (e.g., elimination of general recursion by introducing explicit stacks [28]) to produce dataflow circuits. The associated runtime comprises a garbage collector for immutable Haskell values [2] allocated in on-chip memory.

O2B (*OCaml on Board)*[24] is an implementation of the OCaml VM on a softcore processor. It allocates the OCaml heap in the memory of the softcore processor, either in on-chip memory or in external (SDRAM) memory. O2B can be used in combination with the hardware compiler Macle (*ML accelerator*) for accelerating certain OCaml functions (by compiling them into VHDL). Like our approach, the accelerated code can operate on OCaml data structures. However, the timing behavior of Macle functions is unspecified (the compiler can choose), and synchronizations are not instantaneous, notably for let-bindings. Macle therefore cannot be used for designing synchronous circuits. Macle provides built-in algorithmic skeletons, such as a parallel *map* for OCaml arrays. In the context of our OCaml VM implementation, these skeletons could be directly expressed by the programmer, in Eclat.

Hardware implementations of the Warren Abstract Machine (WAM) [11] have also been proposed. The Archlog system [10] for instance, similarly to our two-level approach, enables the implementer to generate custom VM architectures using a Prolog-like language, to run realistic Prolog applications on them.

[10] https://github.com/janestreet/hardcaml

The design of ECLAT has been influenced by the programming construct *exec* of ESTEREL for calling asynchronous tasks from synchronous code. This programming style has lead to the synchronous reactive language HIPHOP.JS [5], which adds synchronous concurrency and preemption to JAVASCRIPT. A major difference between our work and these approaches is that execution of ECLAT long-running computations is deterministic and fair by construction, since ECLAT is compiled to synchronous circuits, including its computational part.

7 Conclusion

This paper presents ECLAT, a synchronous functional language for programming reactive hardware applications on FPGA. ECLAT unifies interaction and computation in the applications, by expressing both as ML-like functions. These functions are either instantaneous (*e.g.,* a sequential circuit) or non-instantaneous (e.g., a tail-recursion producing a result after several clock cycles). The language features a construct to execute, within an instantaneous function, a computation whose response time is not necessarily known statically: the computation progresses step by step, *i.e.*, makes one step each time the caller is re-executed. The language also has global arrays (implemented in on-chip memory) and parallel let-bindings to exploit coarse-grained parallelism at the circuit level.

The ECLAT synchronous semantics unify the logical clock tick and the global clock of the FPGA; the synthesis tool ensuring that the applications meet the timing constraints of the target. Therefore, the programmer does not need to compute a Worst Case Execution Time (WCET) on the generated code.

To demonstrate the effectiveness of the ECLAT language, we have implemented in it an entire VM and runtime system for the OCAML language. This VM implementation achieves a $\times 45$ speedup vs. another implementation of the OCAML VM, written in C and targeting a softcore processor. It allows the programmer to define ECLAT external functions using the OCAML FFI. This is intended for hardware acceleration purpose, with important speedup possibilities. Moreover, this VM can be embedded in reactive ECLAT applications.

This paper focuses on the practical aspect of our approach (*i.e.*, the programming possibilities) without detailing the technical aspects. Compiling a synchronous language like ECLAT to hardware is challenging: especially for mixing interaction and computation. Furthermore, deep circuitry can be involved in computation-oriented reactive applications, forcing the implementer to consider carefully the space-time trade-off. For this purpose, ECLAT lets the programmer (1) define instantaneous functions (inlined at compile time) to have a better throughput, or (2) annotate certain functions with keyword **rec** (to be shared among all its call sites and introduce a pause), breaking the critical path and reducing the area of the resulting hardware. Certainly, implementing low-level primitives, such as floating-point operations, requires knowledges in hardware design. However, once these primitives are efficiently implemented, they can be called transparently by the declarative programmer using our VM approach, to express complex computations, while interacting with the physical world.

As future work, we plan to support external memory (SDRAM) to allow larger memory footprints. Loading OCaml bytecode from the environment would be useful for dynamically reconfiguring the code of the OCaml VM without having to re-synthesize it on the FPGA. We also plan to use this work for teaching hardware design and the implementation of declarative languages.

References

1. Baaij, C., Kooijman, M., Kuper, J., Boeijink, A., Gerards, M.: Cλash: Structural descriptions of synchronous hardware using Haskell. In: 2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD 2010). pp. 714–721. IEEE (2010)
2. Barker, M., Edwards, S.A., Kim, M.A.: Synthesized In-Bram Garbage Collection for Accelerators with Immutable Memory. In: 2022 32nd International Conference on Field-Programmable Logic and Applications (FPL '22). pp. 47–53. IEEE (2022)
3. Berry, G.: A hardware implementation of pure Esterel. Sadhana **17**, 95–130 (1992)
4. Berry, G., Gonthier, G.: The Esterel synchronous programming language: Design, semantics, implementation. Science of computer programming (SCP) **19**(2), 87–152 (1992)
5. Berry, G., Serrano, M.: Hiphop.js: (a)synchronous reactive web programming. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20). pp. 533–545 (2020)
6. Bourgeat, T., Pit-Claudel, C., Chlipala, A., Arvind: The essence of bluespec: a core language for rule-based hardware design. In: 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20). pp. 243–257 (2020)
7. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.A.: LUSTRE: A declarative language for programming synchronous systems. In: 14th Annual ACM Symposium on Principles of Programming Languages (POPL 1987). vol. 178, p. 188. ACM (1987)
8. Cheney, C.J.: A nonrecursive list compacting algorithm. Communications of the ACM **13**(11), 677–678 (1970)
9. Colaço, J.L., Pagano, B., Pouzet, M.: Scade 6: A formal language for embedded critical software development. In: 2017 International Symposium on Theoretical Aspects of Software Engineering (TASE). pp. 1–11. IEEE (2017)
10. Fidjeland, A., Luk, W.: Archlog: High-level synthesis of reconfigurable multiprocessors for logic programming. In: 2006 International Conference on Field Programmable Logic and Applications. pp. 1–6. IEEE (2006)
11. Fidjeland, A., Luk, W., Muggleton, S.: Scalable acceleration of inductive logic programs. In: 2002 IEEE International Conference on Field-Programmable Technology, 2002.(FPT). Proceedings. pp. 252–259. IEEE (2002)
12. Gammie, P.: Synchronous digital circuits as functional programs. ACM Computing Surveys (CSUR) **46**(2), 1–27 (2013)
13. Gautier, T., Le Guernic, P., Besnard, L.: Signal: A declarative language for synchronous programming of real-time systems. In: Kahn, G. (ed.) Functional Programming Languages and Computer Architecture. pp. 257–277. Springer Berlin Heidelberg, Berlin, Heidelberg (1987)
14. Johnsson, T.: Lambda lifting: Transforming programs to recursive equations. In: Jouannaud, J.P. (ed.) Functional Programming Languages and Computer Architecture. pp. 190–203. Springer Berlin Heidelberg, Berlin, Heidelberg (1985)
15. Kennedy, A.: Compiling with continuations, continued. In: 12th ACM SIGPLAN International Conference on Functional programming (ICFP '07). pp. 177–190 (2007)
16. Krivine, J.L.: A call-by-name lambda-calculus machine. Higher-order and symbolic computation **20**, 199–207 (2007)

17. Leroy, X.: The ZINC experiment: an economical implementation of the ML language. Tech. rep., INRIA (1990)
18. Li, Y., Leeser, M.: HML, a novel hardware description language and its translation to VHDL. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **8**(1), 1–8 (2000)
19. Mycroft, A., Sharp, R.: A statically allocated parallel functional language. In: International Colloquium on Automata, Languages, and Programming (ICALP). pp. 37–48. Springer (2000)
20. Pizani Flor, J.P., Swierstra, W., Sijsling, Y.: Pi-ware: Hardware description and verification in Agda. In: 21st International Conference on Types for Proofs and Programs (TYPES 2015). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2018)
21. Potop-Butucaru, D., De Simone, R., Talpin, J.P.: The synchronous hypothesis and synchronous languages. The embedded systems handbook pp. 1–21 (2005)
22. Rocheteau, F., Halbwachs, N.: Implementing reactive programs on circuits a hardware implementation of LUSTRE. In: Real-Time: Theory in Practice: REX Workshop Mook. pp. 195–208. Springer (1992)
23. Saint-Mleux, X., Feeley, M., David, J.P.: SHard: a Scheme to hardware compiler. In: Workshop on Scheme and Functional Programming (2006)
24. Sylvestre, L., Chailloux, E., Sérot, J.: Accelerating OCaml programs on FPGA. International Journal of Parallel Programming (IJPP) **51**(2-3), 186–207 (2023)
25. Sylvestre, L., Chailloux, E., Sérot, J.: Work-in-Progress: mixing computation and interaction on FPGA. In: 2023 International Conference on Embedded Software (EMSOFT '23). pp. 5–6. IEEE (2023)
26. Townsend, R., Kim, M.A., Edwards, S.A.: From functional programs to pipelined dataflow circuits. In: 26th International Conference on Compiler Construction (CC '17). pp. 76–86 (2017)
27. Varoumas, S., Pesin, B., Vaugon, B., Chailloux, E.: Programming microcontrollers through high-level abstractions: The OMicroB project. Journal of Computer Languages (COLA) **77**, 101228 (2023)
28. Zhai, K., Townsend, R., Lairmore, L., Kim, M.A., Edwards, S.A.: Hardware synthesis from a recursive functional language. In: 2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS 2015). pp. 83–93. IEEE (2015)