Accelerating OCaml Programs on FPGA

Loïc Sylvestre, Emmanuel Chailloux, Jocelyn Sérot

International Journal of Parallel Programming manuscript No.
(will be inserted by the editor)

Accelerating OCaml programs on FPGA

Loïc Sylvestre · Emmanuel Chailloux · Jocelyn Sérot

Received: date / Accepted: date

Abstract This paper aims to exploit the massive parallelism of Field-Programmable Gate Arrays (FPGAs) by programming them in OCaml, a multiparadigm and statically typed language. It first presents O2B, an implementation of the OCaml virtual machine using a softcore processor to run the entire OCaml language on an FPGA. It then introduces Macle, a language to express, in ML-style, hardware-accelerated user-defined functions, implemented as gates and registers on the same FPGA. Macle allows to implement pure computations and compose them in parallel. It also supports processing dynamic data structures such as arrays, matrices and trees allocated by the OCaml runtime in the memory of the softcore processor. Macle functions can then be called, as hardware accelerators, by OCaml programs executed by O2B. This combination of Macle and OCaml codes in a single source program enables to easily prototype FPGA applications mixing numeric and symbolic computations.

Keywords High-level parallel programming, FPGA, OCaml, Virtual machine, Hardware acceleration, Compiling

1 Introduction

Reconfigurable circuits, like Field-Programmable Gate Arrays (FPGAs), are suited to design custom architectures exploiting the concurrent nature of hardware structures [6]. The configuration of an FPGA is commonly produced by a synthesis toolchain from a description expressed in hardware description language (HDL) such as VHDL or Verilog. Other examples of more expressive HDLs include Chisel [4]

Loïc Sylvestre · Emmanuel Chailloux
Sorbonne Université, CNRS, LIP6, F-75005 Paris, France
E-mail: Loic.Sylvestre@lip6.fr, Emmanuel.Chailloux@lip6.fr

Jocelyn Sérot
Université Clermont Auvergne, Clermont Auvergne INP, CNRS, Institut Pascal, F-63000 Clermont-Ferrand, France
E-mail: Jocelyn.Serot@uca.fr

embedded in Scala, Clash [3] in Haskell, MyHDL [10] in Python and HardCaml[1] in OCaml. Nevertheless, the *Register Transfer Level* (RTL) programming model, on which HDLs are based, is characterized by a very low level of abstraction. Hence, different approaches aim to hardware-accelerate software applications using FPGAs.

- There have been some attempts to compile small applicative languages, such as SHard [23], FLOH [27] and Basic SCI [14], directly to RTL [13]. A representative example is SAFL (*Statically Allocated Functional Language*) [20], which is a first-order ML-like language limited to tail recursion and static data structures.
- For more complex languages, custom processors or virtual machines can be implemented in RTL to run high-level languages on FPGA. JAIP [28] is a Java Virtual Machine (JVM) written in VHDL, calling a softcore processor[2] to handle dynamic class-loading. JikesRVM [19] is a JVM implemented on a CPU using an FPGA for accelerating automatic dynamic memory management.
- High-Level Synthesis (HLS) promotes the use of imperative languages to design hardware [21]. Most HLS tools, such as Catapult C or Handel-C, support a subset of C annotated with pragmas to optimize the compilation to RTL. LegUp [5] runs C programs on a softcore processor while compiling functions – those that do not use dynamic allocation and recursion – to RTL. Pylog [15] proposes a similar approach for running Python on FPGA platforms having a hardcore processor.
- Other HLS tools[3] use OpenCL to express parallel applications and target heterogenous architectures involving Multicores, GPUs and FPGAs. Aparapi [24] and GVM [12] implement the JVM in OpenCL. TAPA [7] is a framework for task parallelism targeting OpenCL. TornadoVM [22] compiles specially annotated Java code to OpenCL. These tools, however, do not sufficiently expose the fine-grained parallelism available on the FPGAs nor their customization possibilities.
- FPGAs allows implementing parallel programming models [18] like task-parallelism [7] and parallel skeletons [9]. For instance, Lime [2] is a task-based data-flow programming language compiled to OpenCL or Verilog and interacting with Java bytecode running on a CPU. Kiwi [25] is a subset of C♯ compiled to RTL and offering events, monitors and threads. RIPL [26] is an image processing language with a collection of parallel skeletons.

The work described in this paper builds on the results of these experiments, by proposing an approach in which:

- a runtime system, implemented on a softcore processor, is used to allow high-level programming on FPGA (like JAIP);
- hardware acceleration of user-defined functions (like SAFL) is provided by partitioning the application code between the host and the accelerated code (like Pylog);
- the host language, running the softcore processor, and the embedded language used to describe accelerated functions are similar (like Lime);
- some predefined parallel constructs are provided to ease exploitation of the massive parallelism offered by FPGAs (like Kiwi).

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

[2] A softcore processor is processor implemented in the reconfigurable part of an FPGA.

[3] Such as AMD Vivado HLS and Intel OpenCL SDK.

To this end, we have:

1. ported the OCaml virtual machine (VM) and its runtime (including a garbage collector) on a softcore processor to support the entire OCaml language.
2. combined this VM approach with hardware acceleration of user-defined functions expressed in an ML-like language. This language is extended with parallelism skeletons in order to process dynamic data structures allocated by the OCaml runtime in the memory of the softcore processor.

This approach allows to take full advantage of the fine-grained parallelism of FPGAs, while programming them in OCaml, and hence supporting quick prototyping, static type-checking, simulation and debugging of applications mixing numeric and symbolic computations.

Our contributions are:

- O2B[4] (*OCaml On Board*), a port of OMicroB [29] (an implementation of the OCaml Virtual Machine) targeting the Nios II softcore processor realized on an FPGA. O2B enables to call custom hardware accelerators from OCaml programs.
- Macle[5] (*ML accelerator*), a subset of OCaml designed to express hardware-accelerated user-defined functions, called *Macle circuits*. These functions – distinguished with a special keyword "**circuit**" – are compiled to VHDL and synthetised on an FPGA to be used as hardware accelerators from OCaml source programs executed by O2B. Glue code is automatically generated. It includes C and OCaml code, VHDL descriptions and scripts to control the end-to-end synthesis workflow. Macle supports OCaml data structures (such as lists, trees, arrays and matrices) allocated in the OCaml VM heap. Macle allows general recursion by automatically rewriting it into tail-recursion with a local stack implemented in on-chip memory. Finally, Macle provides parallelism skeletons over OCaml arrays to expose fine-grained parallelism and optimize memory transfers.

The remainder of this paper is organized as follows. Sect. 2 introduces the O2B infrastructure to run OCaml programs on an FPGA. Sect. 3 proposes an approach to accelerate OCaml programs augmented with Macle circuits. Sect. 4 presents the compilation of Macle, using an intermediate language (HSML, *Hierarchical State Machine Language*) to abstract the VHDL target. Sect. 5 evaluates our approach on different benchmarks to measure the speedup resulting from using hardware-acceleration in Macle. Sect. 6 describes a mechanism using parallel skeletons to optimize memory transfers when accessing the OCaml heap. Sect. 7 discusses the acceleration possibilities and the programming style obtained and then identifies future work.

2 The O2B framework

O2B (*OCaml On Board*) is a tool to run OCaml programs on FPGAs. It is based on OMicroB [29], an implementation of the OCaml VM dedicated to high-level programming of microcontrollers with scarce resources.

[4] https://github.com/jserot/O2B
[5] https://github.com/lsylvestre/macle

2.1 Compilation flow for OCaml to FPGAs

Fig. 1 describes the configuration process used to run OCaml programs on an Intel FPGA[6] via O2B. The bytecode generated by the OCaml compiler is transformed into a static C array, then embedded in the C program implementing the bytecode interpreter and the O2B runtime library, including a garbage collector (GC). The OCaml heap and stack are C static arrays. This program is associated with the functions of the Board Support Package (BSP) giving access to the hardware resources of the target board. The resulting application is compiled to binary code executable by the Nios II softcore processor.

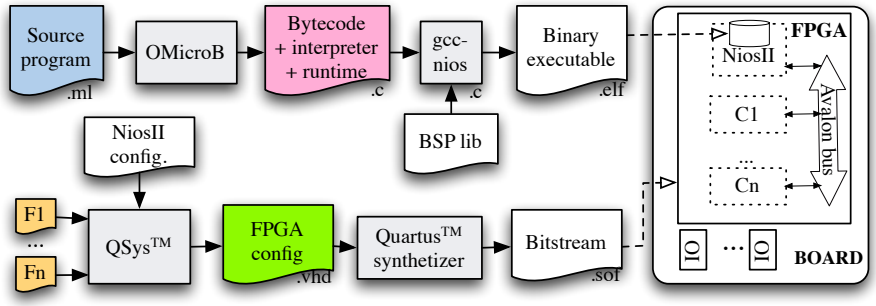


Fig. 1 Compilation flow targeting Intel FPGAs

The complete FPGA configuration includes the exact architecture of the processor used as well as a set of external RTL descriptions F1 ⋯ Fn to be implemented as *custom components* C1 ⋯ Cn. Technically, this configuration step is carried out by the QSys tool of the Intel Quartus chain. It generates a set of VHDL files which constitutes the description of the hardware platform. This description includes the components C1 ⋯ Cn and the Nios II processor to be synthesized through the Quartus chain to reconfigure the FPGA.

The OCaml heap and stack can be stored either in the on-chip memory of the target FPGA (for small programs) or in the external memory (SDRAM) of the board. In both cases, access is provided by means of an interconnection bus[7]. This bus also supports data transfers between the custom components and the binary code executed by the processor. Both the softcore and the custom components can access the physical IOs of the FPGA.

[6] This process is general and can be adapted to target other FPGA families.

[7] Avalon bus for Intel platforms.

2.2 Calling accelerators from OCaml programs

The OCaml language offers an OCaml/C foreign function interface (FFI) to call C functions from OCaml programs. These C functions, running on the softcore, can in turn invoke custom components implemented on the FPGA. It is thus possible to use custom components from OCaml programs compiled to bytecode executed by O2B. The communication layer between O2B and a custom component is done via a set of dedicated registers associated to the component and mapped into the memory of the softcore processor.

Fig. 2 shows the source code of an OCaml program designed to run with O2B. It defines three implementations of the *greatest common divisor* (GCD) algorithm: one in OCaml using a tail recursion, one in C using a loop, and one in C using an hardware accelerator that will be defined at Fig. 7 in VHDL.

```
external gcd_c : int -> int -> int ;;
external gcd_rtl : int -> int -> int ;;

let rec gcd_caml a b =
  if a > b then gcd_caml (a-b) b else
  if a < b then gcd_caml a (b-a)
           else a ;;

let chrono f a b =
  let t1 = Timer.get_us () in
  let res = f a b in
  let t2 = Timer.get_us () in
  print_int (t2-t1) ;;

let main() =
  Timer.init () ;
  let a = 5000 and b = 7000 in
  chrono gcd_caml a b ;
  chrono gcd_c a b ;
  chrono gcd_rtl a b ;;

main () ;;
```

(a) OCaml program

```
value gcd_c(value m, value n){
  int a, b;
  a = Int_val(m);
  b = Int_val(n);
  while ( a != b ) {
    if ( a > b ) a = a-b;
    else b = b-a;
  }
  return Val_int(b);
}

value gcd_rtl(value m, value n){
  int res;
  GCD_ARG(0,Int_val(m));
  GCD_ARG(1,Int_val(n));
  GCD_START();

  while (! GCD_RDY())
    ;
  res = GCD_RESULT();
  return Val_int(res);
}
```

(b) external C code

Fig. 2 An OCaml program using external C code on O2B

The difference between two calls to `Timer.get_us` (before and after a computation) in the OCaml function `chrono` gives the execution time of the argument function call in microseconds. The C function `printf`, and by extension, the OCaml functions `print_int` and `print_string` use the Board Support Package of the FPGA target to write on a console[8]. The `gcd_c` and `gcd_rtl` functions are defined as external functions in the OCaml code using the standard FFI mechanism. Calling a custom component from the `gcd_rtl` function involves sending the argument to and retrieving the result from the dedicated registers of the custom component. The corresponding operations are abstracted by the C macros `GCD_ARG`, `GCD_START`, `GCD_RDY`

[8] The FPGA board is connected to a host PC via an UART connection for printing and debugging.

and `GCD_RESULT`). The behavioral description of GCD in VHDL requires 50 lines of code, plus 100 lines of VHDL glue code for argument and result passing. Finally, this GCD component must be mapped into the global configuration of the system implemented on the FPGA (called the *System on Programmable Chip*, SoPC), either manually (using the QSys tool) or by scripting. In practice, this limits the use of the O2B framework to programmers familiar both with VHDL and the target toolchain (Intel Quartus in the current distribution).

In the rest of the paper, we describe how this limitation can be overcome by providing a ML-like language, similar to OCaml, to describe hardware-accelerated functions and automatically generate both the FPGA configuration and glue code to interact with OCaml programs running on the softcore.

3 High-level FPGA programming

The proposed approach relies on a dedicated ML-like language to express hardware-accelerated functions. This language, called Macle (*ML Accelerator*), can interoperate with the OCaml runtime of O2B and therefore can be used to accelerate OCaml *host* programs running on a softcore processor realized on an FPGA.

3.1 Compilation Flow

Fig. 3 shows our compilation flow of OCaml to FPGA. It automatically generates the configuration of an FPGA from an OCaml program extended with hardware-accelerated functions defined in Macle. The OCaml code is compiled to bytecode to be executed by O2B targeting a softcore processor implemented on the FPGA whereas each Macle circuit is compiled to VHDL and then synthesized as a custom hardware component usable from the OCaml program. The glue code (including OCaml, C and VHDL files) is automatically generated from inferred types of Macle circuits. The compilation flow can therefore be used by a programmer without prior knowledge in VHDL or experience with the target FPGA programming toolset.

In Fig. 3, the two double arrows denote the calls of C functions from OCaml and the use of VHDL code from C.

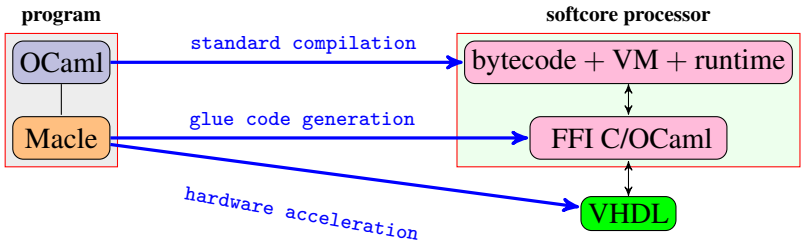


Fig. 3 Accelerating OCaml programs on FPGA using O2B and Macle

3.2 The Macle language

Macle is a ML-like language which includes:

- a functional-parallel core language (called Macle Core) compiled to RTL;
- additional language constructs (implemented in RTL) to interact with the OCaml runtime.

Fig. 4a defines the syntax of Macle Core. This language is independent of OCaml and can be used to program synchronous circuits and compose them in parallel. We denote by $\overrightarrow{o}$ (or $o_1 \cdots o_n$) a non-empty sequence of objects o_i. Macle Core includes variables (taken from a set of name $\mathscr{X}$), constants, applications of builtin operators and conditionals. It also offers local mutually tail-recursive functions, function calls and let bindings. A simple let binding **let** $x = e$ **in** e' first computes e, then e'. By extension, a multiple let-binding **let** $x_1 = e_1$ **and** $\cdots$ $x_n = e_n$ **in** e' first computes the expressions $e_1 \cdots e_n$ in parallel and synchronizes before computing e'. For instance, the hardware implementation of (**let** $x =$ factorial 10 **and** $y =$ factorial 11 **in** $x+y$) instantiates twice the implementation of factorial function in order to enable their parallel execution. Function calls use an implicit parallel let-binding to compute the arguments passed to each function. Non-recursive functions can take functions as arguments[9].

circuit	$ci ::=$ **circuit** $f\ x_1 \cdots x_n = e$
constant	$c ::= \langle bool \rangle \mid \langle integer \rangle \mid ()$
variable	$x, y, f \in \mathscr{X}$
operator$_1$	$\ominus ::= - \mid$ **not** $\mid \cdot\cdot$
operator$_2$	$\oplus ::= + \mid < \mid \cdot\cdot$
expression	$e ::= x \mid c \mid \ominus e \mid e_1 \oplus e_2$
	$\mid$ **if** e **then** e_1 **else** e_2
	$\mid e_1 \mid\mid e_2$
	$\mid e_1$ && e_2
	$\mid$ **let** $f\ \overrightarrow{x_1 \cdots x_n = e}$ **in** e'
	$\mid$ **let rec** $f\ \overrightarrow{x_1 \cdots x_n = e}$ **in** e'
	$\mid f\ e_1 \cdots e_n$
	$\mid$ **let** $x_1 = e_1$ **and** $\cdots$ $x_n = e_n$ **in** e'
	$\mid \cdot\cdot$

(a) Macle Core

exception	$exn ::=$ `Failure` $\langle string \rangle$
	$\mid$ `Stack_overflow`
	$\mid \cdot\cdot$
pattern	$p ::= C(x_1, \cdots x_n)$
	$\mid C$
expression	$e ::= \cdot\cdot$
	$\mid$ **raise** exn
	$\mid$ **match** e **with** $\overrightarrow{p \to e'}$
	$\mid$ **ref** e
	$\mid\ !\, e$
	$\mid e := e'$
	$\mid e.(e')$
	$\mid e.(e') \leftarrow e''$
	$\mid$ **array_length** e
	$\mid e\ ;\ e'$
	$\mid$ **for** $x = e$ **to** e' **do** e'' **done**

(b) Interaction with OCaml

Fig. 4 Syntax of the Macle language

Fig. 4b defines the syntax of the Macle subset interacting with the OCaml runtime. It comprises:

[9] Each call of these functions is specialized and inlined at compile time.

- **raise** *exn* for raising a built-in exception *exn* such as `Failure` (parametrized by a literal strings) or `Stack_overflow`;
- **match** $\cdots$ **with** $\cdots$ for destructuring (*i.e.*, non-nested pattern matching) values of an algebraic datatype;
- $!e$ for accessing the content of the reference e;
- $e := e'$ for setting the content of the reference e to the value of e';
- $e\,.\,(e')$ for accessing at the index e' of the array e;
- $e\,.\,(e') \leftarrow e''$ for setting the value of e'' at the index e' of the array e;
- **array_length** e for accessing the length of the array e;
- $e\,;\,e'$ which is a syntactic sugar for **let** $x = e$ **in** e' where x is a fresh name;
- **for** $x = e$ **to** e' **do** e'' **done** which is a syntactic sugar for a tail-recursive formulation using *let rec*.

Note that, currently, Macle circuits cannot allocate data structures; they can only manipulate values allocated by the VM in the OCaml heap.

To preserve the semantics and the safety of the Macle code, multiple let-bindings are sequentialized when they contain memory accesses or raise an exception.

General recursion is supported via a program transformation producing code containing only tail-recursive calls and using an explicit stack. When the stack overflows, an exception (`Stack_overflow`) is raised. Recursion in Macle uses an explicit call stack, as described in Sect. 5. Tail-recursion does not require a stack.

Fig. 5a shows three Macle circuits. The circuit `gcd_rtl` expresses the GCD algorithm in Macle Core. The circuit `collatz` computes the *stopping time* of a Collatz [16] sequence (also called *Syracuse*) starting from a given integer. The circuit `sum_array` sums the elements of a given OCaml array.

```
circuit gcd_rtl m n =
  let rec gcd a b =
    if a > b then gcd (a-b) b else
    if a < b then gcd a (b-a)
             else a
  in gcd m n ;;

circuit collatz n =
  let rec next len u =
    if u <= 1 then len else
    if u mod 2 == 0
    then next (len+1) (u/2)
    else next (len+1) (3*u+1)
  in next 0 n ;;

circuit sum_array a =
  let n = array_length a in
  let rec loop acc i =
    if i >= n then acc else
    loop (acc + a.(i)) (i+1)
  in loop 0 0 ;;
```

(a) Examples of Macle circuits

```
type exp =
  | Int of int
  | Var of int
  | Add of exp * exp ;;

circuit eval_exp env e =
  let rec eval e =
    match e with
    | Int(n) -> n
    | Var(k) -> env.(k)
    | Add(e1,e2) ->
        eval e1 + eval e2
  in eval e ;;

let main() =
    let env = [|100|] in
    let e = Add(Int(1),Var(0)) in
    try  print_int (eval_exp env e)
    with Failure s -> print_string s ;;

main() ;;
```

(b) Example of program

Fig. 5 Examples of Macle circuits and OCaml program using a Macle circuit

Fig. 5b shows an OCaml program calling a Macle circuit. It allocates an abstract syntax tree in the OCaml heap and evaluates it using the `eval_exp` Macle circuit. Accesses to the OCaml heap are safe since the exception `Failure` is (implicitly) raised in case of an out-of-bounds index or a non-exhaustive pattern matching. This exception can then be caught in OCaml by the **try** ··· **with** construct. In this example, the program evaluates the expression `Add(Int(1),Var(0))` recursively and prints the result. Evaluating `Var(0)` fetches the value at the index 0 of the array.

4 Compiling Macle

The global compilation flow from Macle to VHDL is depicted Fig. 6. It involves four passes. The first pass (1) consists in normalizing the source code:

- renaming all bindings in the source code with unique names;
- rewriting the code in so-called *Administrative Normal Form* [17] (introducing let-bindings for each step of computation);
- inlining functions by recursively duplicating their body at each call site (except recursive ones);
- transforming recursive functions which are not tail-recursive into tail-recursive ones using an explicit stack.

The second pass (2) compiles Macle into an intermediate language, called HSML (*Hierarchical State Machine Language*), allowing to express parallel composition of hierarchical finite state machines. The third pass (3) flattens the hierarchical structure of HSML. The fourth pass (4) translates a flat HSML description into VHDL. At each point of the compilation flow, an OCaml backend is provided for simulation and debugging on a PC, as indicated by the dotted arrows.

The rest of this section focuses on pass 2 and is restricted to Macle Core.

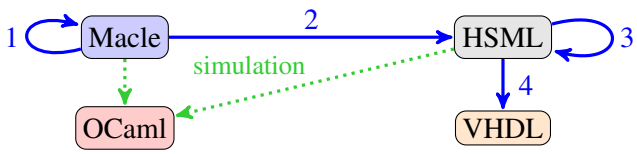


Fig. 6 Compilation flow of Macle to VHDL

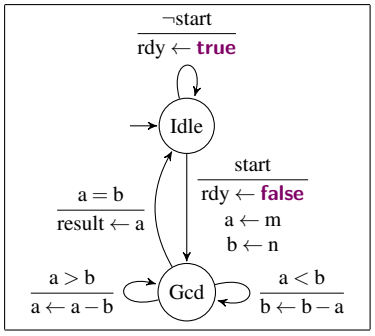
4.1 Targeting the register transfer level

Synchronous finite state machines (FSM) are commonly used to describe computations at the register transfer level (RTL). An FSM is classically defined by a set of states (names) and a set of transitions. Each transition connects a source state to a destination state and can be associated to a set of guards and a set of actions. Guards define when the transition is enabled. They can depend on inputs and local variables.

Actions are performed when the transition is enabled and can write outputs and local variables. Transitions are only taken at the rising edge of a global clock. At each clock edge, if a transition starting from the current state has all its guards validated, it is enabled, the associated actions are performed (instantaneously) and the destination state becomes the current state.

FSMs are classically encoded in VHDL as synchronous processes with asynchronous reset. Inputs, outputs and local variables are implemented as VHDL signals with a dedicated signal representing the current state. At each rising edge of the input clock, depending on the value of the current state and some conditions involving inputs and local variables, the next state value is selected and the value of outputs and local variables is updated. The FSM is re-initialized, asynchronously, whenever the reset input signal becomes true.

Fig. 7 gives a graphical representation of an FSM describing the computation of GCD and its encoding in VHDL. The `start` input and `rdy` output are used respectively to start and signal the termination of the computation. In the VHDL code, modifications of the state variable `STATE` as well as the outputs and local variables use signal assignments (`<signal_name> <= <expression>`). Assignments occurring at the same clock edge are performed concurrently, *i.e.*, the expressions denoted by the right hand sides (RHSs) are all evaluated in parallel and then, and only then, the signals designated by the left hand sides (LHSs) are updated simultaneously.




```
entity gcd_rtl is
  port(signal clk, reset : in std_logic;
       signal start : in std_logic;
       signal rdy : out std_logic;
       signal m, n : in signed(30 downto 0);
       signal result : out signed(30 downto 0));
end entity;

architecture rtl of gcd_rtl is
  type t_state is (Idle, Gcd);
  signal STATE : t_state;
  signal a, b : signed(30 downto 0);
begin process(reset,clk) begin
    if reset = '1' then
      STATE <= Idle;
    elsif rising_edge(clk) then
      case STATE is
        when Idle =>
          if start then
            rdy <= false;
            a <= m;
            b <= n;
            STATE <= Gcd;
          else
            rdy <= true;
            STATE <= Idle;
          end if;
        when Gcd =>
          if a > b then
            a <= a - b;
            STATE <= Gcd;
          elsif a < b then
            b <= b - a;
            STATE <= Gcd;
          else
            result <= a;
            STATE <= Idle;
          end if;
      end case;
    end if
  end process;
end architecture;
```

Fig. 7 FSM and VHDL implementation of the GCD algorithm (given in Macle Fig. 5)

Note that in the code given Fig. 7, arguments and result are encoded as 31-bit signed integers to have the same representation of OCaml value than in the O2B runtime. This enhances the interoperability between Macle and OCaml.

By declaring separate processes – each encoding a given FSM – within the same VHDL architecture, it is easy to implement synchronous parallel composition of FSMs. Each FSM is triggered by the same global clock and has access to the signals declared in the architecture. However, these signals can only be shared for reading as a signal written by a process cannot be written by another process.

4.2 An FSM-based intermediate language

We do not compile Macle circuits directly to VHDL. Instead, we use an intermediate language, HSML (*Hierarchical State Machine Language*) for describing the behavior of FSMs and expressing their composition, and which can be easily translated to VHDL. HSML is inspired by well-known FSM-based formalisms and languages with notions of hierarchy and compositionality, such as Statecharts [11], Communicating Hierarchical State Machines [1] and Lustre-like languages with automata [8].

Fig. 8 defines the syntax of HSML. A circuit is a parallel composition of FSMs $(A_1 \parallel \cdots A_n)$ depending on inputs, modifying outputs and using local variables. An FSM is a set of zero or more mutually recursive transitions in the scope of a body used to initialize it. A transition is a thunk $f() = A$ associating a name f to an FSM A. HSML offers a notion of hierarchy. For instance, given a transition t and an output x, the FSM $(\textbf{do}\ x \leftarrow 0\ \textbf{then}\ (\textbf{let rec}\ t\ \textbf{in}\ f()))$ is a hierarchical formulation of the FSM $(\textbf{let rec}\ t\ \textbf{in}\ (\textbf{do}\ x \leftarrow 0\ \textbf{then}\ f()))$.

circuit	$\phi ::= \textbf{circuit}\ f\ \overrightarrow{x_{in}}\ \textbf{returns}\ \overrightarrow{x_{out}} = \textbf{var}\ \overrightarrow{x}\ \textbf{in}\ P$
parallel composition	$P ::= A_1 \parallel \cdots A_n$
FSM	$A ::= \textbf{let rec}\ t_1\ \textbf{and}\ \cdots\ t_n\ \textbf{in}\ A_{\text{init}}$ $\mid \textbf{if}\ e\ \textbf{then}\ A_1\ \textbf{else}\ A_2$ $\mid \textbf{do}\ x_1 \leftarrow e_1\ \textbf{and}\ \cdots\ x_n \leftarrow e_n\ \textbf{then}\ A$ $\mid f()$ $\mid P\ \textbf{in}\ A$
transition	$t ::= f() = A$
expression	$e ::= x \mid c \mid \ominus\ e \mid e_1 \oplus e_2$
operator$_1$	$\ominus ::= \cdot\cdot$
operator$_2$	$\oplus ::= \cdot\cdot \mid \wedge \mid \vee$

Fig. 8 Syntax of HSML

An FSM A is a mutually recursive definition of transitions $(\textbf{let}\ \cdots\ \textbf{rec}\ \cdots)$, a conditional, an assignment $(\textbf{do}\ \cdots\ \textbf{then}\ \cdots)$, a branch $f()$ to a transition $f() = A$ or a local parallel composition of FSM $(A_1 \parallel \cdots A_n)\ \textbf{in}\ A$. An assignment $\textbf{do}\ x_1 \leftarrow e_1\ \textbf{and}\ \cdots\ x_n \leftarrow e_n\ \textbf{in}\ A$ evaluates the expressions $e_1\ \cdots\ e_n$, then assigns the results

to the variables $x_1 \cdots x_n$ and finally computes A. An expression e is a variable, a constant or the application of a built-in operator. Logical operators $\wedge$ and $\vee$ are strict.

Fig. 9 shows an HSML circuit corresponding to the VHDL code given Fig. 7. This circuit was automatically generated from the Macle circuit `gcd_rtl` defined Fig. 5.

```
circuit gcd_rtl (start, m, n) returns (rdy, result) = var a, b in
  let rec idle() =
    if start then (do rdy ← false and a ← m and b ← n then gcd()) else
    (do rdy ← true then idle())
  and gcd() =
    if a > b then (do a ← (a-b) then gcd()) else
    if a < b then (do b ← (b-a) then gcd()) else
    (do result ← a then idle())
  in
    (do rdy ← true then idle())
```

Fig. 9 HSML circuit implementing the GCD algorithm

HSML exposes the semantics of the register transfer level (RTL, informally presented on the VHDL code of Fig. 7) while allowing hierarchical formulations that makes it close to an expression language. In particular, some HSML constructs (such as *let rec* and conditional) are common with Macle. Therefore, HSML constitutes a useful intermediate language for compiling Macle to VHDL.

4.3 Compiling Macle Core to HSML

The compilation $\mathscr{C}[\![\textbf{circuit } f\ \overrightarrow{x} = e]\!]$ of a Macle Core circuit is defined as the compilation of the body e of the circuit, from which the inputs, outputs and local variables are inferred.

$$\mathscr{C}_{\mathsf{ci}}[\![\textbf{circuit } f\ \overrightarrow{x} = e]\!] = \textbf{circuit } f\ \overrightarrow{x_{in}}\ \textbf{returns}\ \overrightarrow{x_{out}} = \textbf{var}\ \overrightarrow{x_{local}}\ \textbf{in}\ \overbrace{\mathscr{C}[\![e]\!]^{start,rdy,result}}^{A}$$

$$\text{where} \begin{cases} \overrightarrow{x_{in}}, \overrightarrow{x_{out}} \text{ and } \overrightarrow{x_{local}} \text{ are inputs, outputs and local} \\ \text{variables declarations inferred from } A \\ start, rdy, result \text{ are fresh names} \end{cases}$$

The compilation $\mathscr{C}[\![e]\!]^{start,rdy,result}$ of a Macle Core expression e is a hierarchical FSM initialized in a special state *idle*. It waits for the input *start* to be set to the value **true** to start the computation. This computation assigns a value to the output *result*. The output *rdy* notifies when the computation is done. The auxiliary function $\mathscr{C}_{\mathsf{e}}[\![e]\!]_{\rho}^{result,idle}$ is defined next. The compilation environment ρ maps functions names to the list of their formal arguments.

$$\mathscr{C}[\![e]\!]^{start,rdy,result} = \left(\begin{array}{l} \textbf{let rec } idle() = \\ \quad \textbf{if } start \textbf{ then } (\textbf{do } rdy \leftarrow \textbf{false then } \mathscr{C}_{\mathsf{e}}[\![e]\!]_{\emptyset}^{result,idle}) \\ \quad \textbf{else } (\textbf{do } rdy \leftarrow \textbf{true then } idle()) \\ \textbf{in } (\textbf{do } rdy \leftarrow \textbf{true then } idle()) \end{array} \right)$$

where *idle* is a fresh name

The compilation $\mathscr{C}_{\mathsf{e}}[\![e]\!]_{\rho}^{r,idle}$ of a subexpression is inductively defined on the syntax of the expressions. The compilation of a subexpression e which does not contain control structures is defined as an affectation of e to a variable r continuing with a tail-call to a destination.

$$\mathscr{C}_{\mathsf{e}}[\![e]\!]_{\rho}^{r,idle} = \textsf{do } r \leftarrow e \textsf{ then } idle()$$
if e is a variable, a constant or an application of an operator

The compilation of a Macle conditional is an HSML conditional, subexpressions being inductively compiled.

$$\mathscr{C}_{\mathsf{e}}[\![\textsf{if } x \textsf{ then } e_1 \textsf{ else } e_2]\!]_{\rho}^{r,idle} = \textsf{if } x \textsf{ then } \mathscr{C}_{\mathsf{e}}[\![e_1]\!]_{\rho}^{r,idle} \textsf{ else } \mathscr{C}_{\mathsf{e}}[\![e_2]\!]_{\rho}^{r,idle}$$

Compiling a *let rec* globalizes function parameters. To achieve this, each function name introduced by a *let rec* is bound to the list of its formal parameters within the compilation environment ρ. The extension of ρ with a function name f bound to its parameters $x_1 \cdots x_n$ is denoted by $\rho[f \mapsto (x_1, \cdots x_n)]$, assuming that f is not in the domain of ρ. Alternatively, the compilation of a function call $(f\ x_1 \cdots x_n)$ is an assignment of the values $x_1 \cdots x_n$ to the formal parameters $y_1 \cdots y_n$ given by $f(\rho)$, continuing with a call to $f()$.

$$\mathscr{C}_{\mathsf{e}}\left[\!\!\left[\begin{array}{l}\textsf{let rec } f_1\ \overrightarrow{x_1} = e_1\\ \textsf{and } \cdots\ f_n\ \overrightarrow{x_n} = e_n\\ \textsf{in } e\end{array}\right]\!\!\right]_{\rho}^{r,idle} = \left(\begin{array}{l}\textsf{let rec } f_1\ () = \mathscr{C}_{\mathsf{e}}[\![e_1]\!]_{\rho'}^{r,idle}\\ \textsf{and } \cdots\ f_n\ () = \mathscr{C}_{\mathsf{e}}[\![e_n]\!]_{\rho'}^{r,idle}\\ \textsf{in } \mathscr{C}_{\mathsf{e}}[\![e]\!]_{\rho'}^{r,idle}\end{array}\right)$$
where $\rho' = \rho[f_1 \mapsto \overrightarrow{x_1}] \cdots [f_n \mapsto \overrightarrow{x_n}]$

$$\mathscr{C}_{\mathsf{e}}[\![f\ x_1 \cdots x_n]\!]_{\rho}^{r,idle} = \textsf{do } y_1 \leftarrow x_1 \textsf{ and } \cdots y_n \leftarrow x_n \textsf{ then } f()$$
if $\rho(f) = (y_1, \cdots y_n)$

The compilation $\mathscr{C}_{\mathsf{e}}[\![\textsf{let } x = e \textsf{ in } e']\!]_{\rho}^{r,idle}$ of a *let* with a single binding is defined as the compilation of the subexpression e into the variable x continuing with the compilation of the body e'.

$$\mathscr{C}_{\mathsf{e}}[\![\textsf{let } x = e \textsf{ in } e']\!]_{\rho}^{r,idle} = \textsf{let rec } f() = \mathscr{C}_{\mathsf{e}}[\![e']\!]_{\rho}^{r,idle} \textsf{ in } \mathscr{C}_{\mathsf{e}}[\![e]\!]_{\rho}^{x,f}$$
where f is a fresh name

The compilation of a *let* with more than one binding is defined as a parallel composition of FSMs followed by a synchronization barrier activating the execution of the compiled body of the *let*.

$$\mathscr{C}_{\mathsf{e}}\left[\!\!\left[\begin{array}{l}\textsf{let } x_1 = e_1\\ \textsf{and } \cdots\ x_n = e_n\\ \textsf{in } e\end{array}\right]\!\!\right]_{\rho}^{r,idle} = \left(\begin{array}{l}\textsf{let rec } f() =\\ \quad \textsf{do } start_1 \leftarrow \textsf{false and } \cdots\ start_n \leftarrow \textsf{false}\\ \quad \textsf{then}\\ \quad ((\mathscr{C}[\![e_1]\!]^{start_1, rdy_1, x_1} \| \cdots\ \mathscr{C}[\![e_n]\!]^{start_n, rdy_n, x_n}) \textsf{ in}\\ \quad\ \textsf{if } rdy_1 \wedge \cdots\ rdy_n \textsf{ then } \mathscr{C}_{\mathsf{e}}[\![e]\!]_{\rho}^{r,idle} \textsf{ else } f())\\ \textsf{in}\\ \textsf{do } start_1 \leftarrow \textsf{true and } \cdots\ start_n \leftarrow \textsf{true}\\ \textsf{then } f()\end{array}\right)$$
(if $n > 1$)

where $\begin{cases} i \in \{1, \cdots n\} \\ f, start_i, rdy_i \text{ are fresh names} \end{cases}$

Parallel let-bindings provide the main possibilities of acceleration of OCaml programs on FPGA as shown in the next section.

5 Examples and benchmarks

We now evaluate the speedup that can be achieved by running OCaml programs on FPGA via O2B and Macle. These programs are assessed by taking as reference an equivalent C code running on the same softcore processor. We first consider programs using circuits written in Macle Core (defined Fig. 4a) and then Macle circuits interacting with the OCaml runtime (using constructs defined Fig. 4b).

5.1 Methodology

Experimental setup We use a Max10 Intel FPGA embedded on a Terasic DE10-LITE board. This FPGA has limited resources: 50K logic elements (LEs), and 1,638 Kbit of on-chip memory. The board itself has 64 MB of external memory (SDRAM) and a clock frequency of 50 MHz. From a given OCaml source program, O2B creates a C program containing the bytecode generated by the OCaml compiler, the VM, its runtime library (including a GC) and additional C code. The bytecode as well as the OCaml stack and heap are implemented with C static arrays, both stored in external memory. The stack size is of 6,400 words of 4 bytes while the heap size is of 4MB. The resulting C program is compiled via the Nios II backend of `gcc` with optimizations enabled (-Os). The Macle circuits (compiled to VHDL) and the softcore processor are synthetized using Quartus 20.1. All data structures manipulated by OCaml, C and Macle code use the OCaml heap. Bounds of OCaml arrays are dynamically checked at each access.

Measuring elapsed time Macle circuits are called from a C block executed on the softcore. For this, and as described in Sect. 2.2, it is necessary to write arguments in the dedicated registers of the custom component implementing a circuit, start this circuit and wait for the end of the computation to read the result (again in the dedicated registers of the custom component). These reads and writes are done via the Avalon SOPC bus. We measure the execution time of each Macle circuit from the beginning to the end of the corresponding C block. The reported times, therefore, include the time to transfer the arguments and results.

5.2 Macle Core

Pure Computations The throughput of the Macle circuit `gcd_rtl` given Fig. 5 is of exactly one tail-call per clock tick at 50 Mhz (*i.e.* 50 million tail-calls per second). We measure the execution time of `gcd_rtl` compared to that of the C function `gcd_c` (given Fig. 2a) running on the softcore. The observed Macle vs C speedup factor is 28×. The hardware implementation of `gcd_rtl` uses approximately 360 logic elements (LEs), *i.e.* 0.75% of the total number of LEs available on the FPGA. Fig. 10

summarizes these results and gives similar examples of tail-recursive functions expressed in Macle vs C, both called from an OCaml program executed by O2B. This benchmark comprises the greatest common divisor (GCD), the recursive sum of the n-th first positives integers (SUM_INT), The Fibonacci sequence (FIBONACCI), a tail-recursive version of the McCarthy 91 function (F91), and the Collatz sequence (COLLATZ) as defined Fig. 5a in Macle. The key point is that these "pure" Macle circuits have a throughput of one tail-call per clock tick while an iteration in the C code results in a sequence of instructions, hence the speedup of Macle vs C depends on the nature of the computation. Moreover, the size (in LEs) of the hardware generated from Macle also depends on the nature of the computation.

tail-recursive function	GCD	SUM_INT	FIBONACCI	F91	COLLATZ
speedup Macle vs C	28×	28×	42×	42×	60×
size (in LEs)	360	275	335	345	360

Fig. 10 Speedup factor of pure computations defined in Macle vs C along with resource usage

Parallel computations Fig. 11a gives a circuit `sum_gcd2` calling twice a function `gcd` and combining results. The **let** ⋯ **and** ⋯ **in** ⋯ constructs is implemented by a synchronization barrier involving a parallel composition of two instances of the FSM given Fig. 7. The global execution time of the barrier is the max of the execution times of the expressions (`gcd` $\mathtt{a}_i$ `y`), to which is added the execution time of the rest of the computation (here instantaneous). For instance, calling the circuit $\mathtt{sum\_gcd}_2$ with equal arguments a_1 and a_2 doubles the previous 28× speedup reported in Fig. 10.

```
circuit sum_gcd2 a1 a2 y =
  let rec gcd n m =
    if n > m then gcd (n-m) m else
    if n < m then gcd n (m-n)
             else n
  in
  let x1 = gcd a1 y
  and x2 = gcd a2 y in
  (x1 + x2)
```

(a) circuit `sum_gcd2`

$\mathtt{sum\_gcd}_n$	**size (LEs)**
$\mathtt{sum\_gcd}_2$	753
$\mathtt{sum\_gcd}_4$	1,413
$\mathtt{sum\_gcd}_8$	2,828
$\mathtt{sum\_gcd}_{16}$	5,135
$\mathtt{sum\_gcd}_{32}$	9,823

(b) resource usage

Fig. 11 Parallelization of a computation and impact on the size of the generated hardware

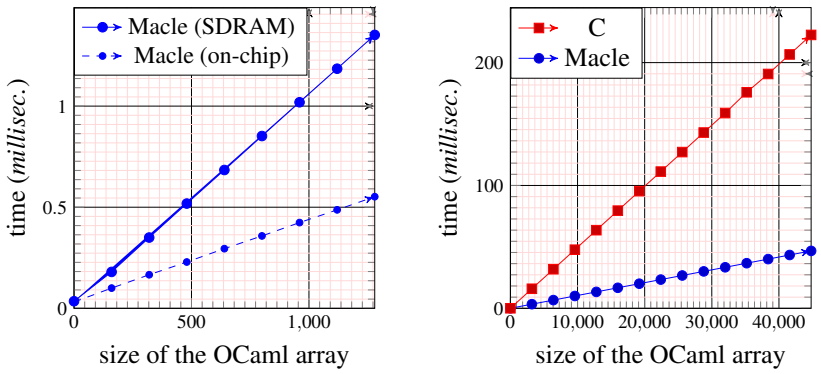
Generalizing this example to circuits $\mathtt{sum\_gcd}_n$ (computing n times `gcd_rtl` and summing the results) gives a speedup of $28 \times n$ in Macle vs C (e.g., $\mathtt{sum\_gcd}_{32}$ is 900 times faster in Macle than in C). This gain is only possible because the `gcd` local function is inlined n times, the generated hardware using more LEs as shown Fig. 11b.

5.3 Interacting with the OCaml runtime

Fig. 12 depicts the execution time of a Macle circuit `sum_array` (given Fig. 5a) computing the sum of the elements of an OCaml array. The size n of the array is its number of elements. The input array is filled with the n first positive integers.

In Fig. 12a, the OCaml heap is limited to 64 KB and is allocated either in on-chip memory or in external memory (SDRAM). Access times are longer (around two times longer on this example) using external memory than on-chip memory. This is still reasonable, considering that this allows to manipulate much larger data structures.

In Fig. 12b, the OCaml heap is allocated in SDRAM (with a heap size of 4MB). The Macle version is 4.8 times faster than the C one. Compared to the speedup obtained on pure computations, this example highlights a bottleneck when using memory accesses from Macle code.



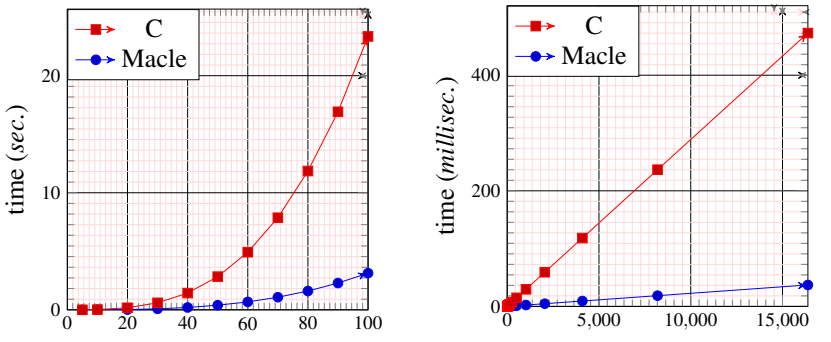
(a) using SDRAM vs on-chip memory (b) using SDRAM

Fig. 12 Execution time of a Macle circuit summing the elements of an OCaml array

Fig. 13a shows the execution time of a Macle circuit `matrix_multiply` multiplying two $n \times n$ matrices filled with positive integers smaller than n, vs a C version. The Macle version (using a classic formulation with three nested loops) is 7.5 times faster than the C one. The generated hardware uses 1,602 LEs.

Fig. 13b shows the execution time of the Macle circuit `eval_exp` (given Fig. 5a) vs a C version, recursively evaluating trees of arithmetic expressions of various sizes (in number of constants and variables). The Macle version is 13 times faster than the C formulation. The realization of this Macle circuit uses 2,461 LEs and an explicit stack of 3,072 words implemented in on-chip memory (using RAM blocks and without requiring bus accesses).

This preliminary evaluation shows that reformulating side-effect-free C functions as Macle circuits can bring substantial speedups (eg., up to 28× for the `gcd_rtl` of Fig. 5). Replicating the hardware corresponding to these circuits, intrinsically resulting in their parallel execution, allows to further boost these speedups (e.g., up to 960× for the $\texttt{sum\_gcd}_{32}$ example given Fig. 10).



(a) circuit `matrix_multiply` (b) circuit `eval_exp`

Fig. 13 Execution time of Macle circuits using imperative features

It also show that for large data structures, such as arrays, the cost of accessing the corresponding memory can quickly create a bottleneck.

6 Optimised transfers and parallel skeletons

As demonstrated in the previous section, allowing Macle circuits to manipulate values stored in the OCaml heap has a cost. Because this heap is implemented in shared memory, each access requires a bus transaction. When manipulating large data structures, like arrays, the corresponding overhead can quickly become prohibitive. To overcome this problem, Macle provides *parallel skeletons* aiming at minimizing this overhead and offering higher-level parallelism. These skeletons are listed Fig. 14.

$$\begin{array}{l}\textsf{array\_map}\langle k\rangle : (\alpha \to \beta) \to \alpha\ \texttt{array} \to \beta\ \texttt{array} \to \texttt{unit} \\ \textsf{array\_reduce}\langle k\rangle : (\alpha \to \beta \to \alpha) \to \alpha \to \beta\ \texttt{array} \to \alpha \\ \textsf{array\_scan}\langle k\rangle : (\alpha \to \beta \to \alpha) \to \alpha \to \beta\ \texttt{array} \to \alpha\ \texttt{array} \to \texttt{unit}\end{array}$$

Fig. 14 Simple parallel skeletons available in Macle

Each skeleton is parameterized by an integer literal k, which statically specifies the size of a buffer used internally to transfer slices of the source and/or destination arrays between the OCaml heap and the Macle circuits:

- ($\textsf{array\_map}\langle k\rangle$ *f src dst*) copies the k first elements of the OCaml array *src* into a VHDL array *buf*, computes the function *f in parallel* on each element of *buf* and writes back the k resulting values in the OCaml array *dst*. Processing the whole OCaml array is carried out by iterating this transfer-execution-transfer sequence.
- ($\textsf{array\_reduce}\langle k\rangle$ *f init a*) sequentialy reduces the OCaml array *a* with the function *f* and an accumulator initialized to *init* by processing array elements k by k. If the body of *f* is a combinatorial expression (*i.e.*, a constant, a variable or an

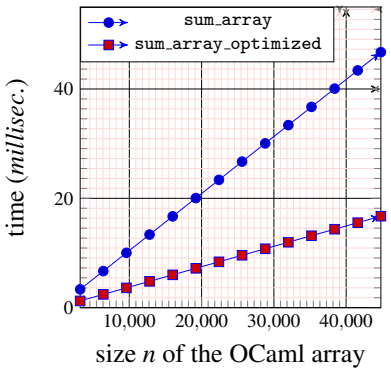
operator applied to combinatorial expressions), each transfer-execution sequence is pipelined thus avoiding the use of a buffer.

– ($\mathbf{array\_scan}\langle k\rangle$ f *init src dst*) sequentialy reduces the OCaml array *src* with the function f and an accumulator initialized to *init* by processing array elements k by k and write back each group of k intermediate steps in the OCaml array *dst*.

Example 1. Fig. 15a illustrates the use of a skeleton **array_reduce**<*k*> to define an optimized version `sum_array_optimized` of the Macle circuit `sum_array` (given Fig. 5a). Fig. 15b gives the execution time of the circuit `sum_array_optimized` vs the circuit `sum_array`. Using **array_reduce**<*k*>, the size of the array is calculated only once rather than at each array accesses. Moreover, since the body of the reduction function f is a combinatorial expression, the execution is piplined and the generated code does not need to use any buffer. The size of the circuit does not depend on k. The `sum_array` and `sum_array_optimized` circuits use respectively arround 570 and 590 LEs. The optimized version is 2.8 times faster than the unoptimized one. Fig. 13a showed that the Macle circuit `sum_array` is 4.8 times faster than C. As a result, there is a speedup of $4.8 \times 2.8 = 13$ when using `sum_array_optimized` vs the C version.

```
circuit sum_array a =
  (* given Fig. 5a *)
  ... ;;


circuit sum_array_optimized a =
  let add n m = n + m in
  array_reduce<4> add 0 a ;;


let main () =
  let n = 14 * 3200 in
  let a = Array.init n (fun x -> x) in
  print_int (sum_array a);
  print_int (sum_array_optimized a) ;;

main ();;
```

(a) Source program



(b) Execution times

Fig. 15 OCaml program with a Macle circuit using the parallel skeleton **array_reduce**<*k*>

Example 2. We consider an OCaml program using a Macle circuit $\texttt{filter\_mul}_k$ replacing all multiples of an integer y by zero in an OCaml array a of size n containing integer from 1 to n. This Macle circuit uses a parallelism skeleton $\mathbf{array\_map}\langle k\rangle$ processing the array a in parallel by slice of k elements. Fig. 16a gives the corresponding program for $k := 64$.

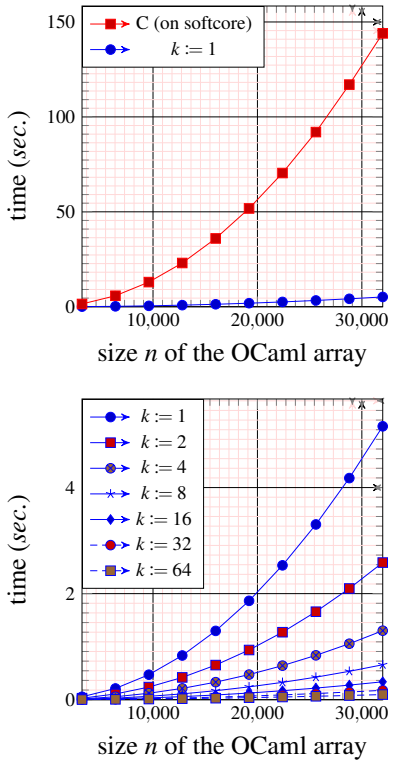
Fig. 16b shows the execution times of the Macle circuit $\texttt{filter\_mul}_k$ for different k vs a C sequential version. Doubling the degree of parallelism k almost doubles both the size of the generated hardware and the speedup (taking into account the transfer time). For instance, when $n = 96{,}000$, $\texttt{filter\_mul}_1$ is 28 times faster than the C version, while $\texttt{filter\_mul}_{64}$ is 60 times faster than $\texttt{filter\_mul}_1$, resulting in a

cumulated speedup of $28 \times 60 = 1{,}680$. This follows a classical space-time trade-off as shown by Fig. 16c, given the sizes (in LEs) of $\texttt{filter\_mul}_k$ for different k.

Comparison with sequential code running on a PC. We assess the performance of the Macle circuit `filter_mul_64` (given Fig. 16a) vs a C sequential version running on a PC equipped with an Intel Core i7 at 2.2 GHz and 16 GB of RAM. This C code is compiled with `gcc` option -Os and called from an OCaml program compiled to native code with the `ocamlopt` compiler. The frequency ratio between the PC and the FPGA is $2.2G/50M = 44$. Due to licensing limitations, the Nios II architecture used in our experiment is a basic, unoptimized one. The sequential C version running on the softcore is $44 \times 8.7 = 380$ slower than the same C code running on the PC. Thus, on this small benchmark, Macle code synthesized on the FPGA is $1{,}680/380 = 4.4$ times faster than an equivalent C sequential code running on a PC.

```
circuit filter_mul_64 y a =

  let rec gcd n m =
    if n > m then gcd (n-m) m else
    if n < m then gcd n (m-n)
             else n
  in

  if y <= 1 then
    raise (Failure "(y > 0) expected")
  else
    let zero_if_mul x =
      if x <= 1 then 0 else
      if x == y then x else
      if gcd x y == 1 then x else 0
    in
    array_map<64> zero_if_mul a a ;;



let interval n =
  Array.init n (fun x -> x + 1) ;;


let main() =
  let n = 32*1000 in
  let a = interval n in
  let y = 2 in

  filter_mul y a ;;

main();;
```

(a) Source program



(b) Execution times

k	1	2	4	8	16	32	65
size (LEs)	1,240	1,655	2,487	4,182	7,521	15,107	29,739

(c) size of `filter_mul` for different k

Fig. 16 OCaml program with a Macle circuit using the parallel skeleton **array_map**<*k*>

7 Conclusion

In this paper, we have proposed an approach for programming FPGAs using the OCaml language. This approach consists in:

- running OCaml programs by embedding their bytecode and the OCaml VM in a C program running on a softcore processor;
- calling hardware-accelerated functions, user-defined in the Macle language from OCaml.

Macle is a functional-imperative subset of OCaml supporting:

- parallel and sequential compositions of computations;
- mixing computations with sequential accesses to the OCaml heap (within the dynamic memory of the softcore processor);
- use of parallelism skeletons on dynamic data structures with optimization of memory transfers.

Hardware acceleration of OCaml functions is simply obtained by replacing a "**let**" keyword in the original OCaml code by "**circuit**". This facilitates porting of OCaml applications, quick prototyping and debugging. Moreover, Macle is a statically typed language that provides much stronger guarantees on the safety of the generated hardware than classical HDLs.

We have presented an implementation of the proposed approach based on the O2B framework augmented with a Macle compiler targeting VHDL. This compilation flow is fully automatized on an Intel FPGA. It is simple to use and includes a simulation mode generating OCaml code from different points of the Macle compiler to test the applications on a PC before synthesizing them on the FPGA. The use of a local stack implemented in on-chip memory (instead of LEs) to realize non-tail recursive Macle functions (as evoked in Sect. 5.3) is a key point to allow large and complex symbolic computations to be implemented on moderately sized FPGAs.

Preliminary results, obtained on small benchmarks are very encouraging. They show in particular that important speedups (up to the three orders of magnitude, compared to C code running on the embedded softcore) can be obtained by combining the ability to compile a Macle function to hardware and the possibility to replicate the corresponding hardware in order to exploit data parallelism. Parametrizable parallel skeletons both offer a manner to address the bottleneck occurring when exchanging data between the OCaml host program and the accelerated Macle functions. It is also a very practical way to explore the space-time trade-off, which constitutes a classical issue when programming FPGAs (reducing computing time by increasing the number of logic elements used).

The work described in this paper offers many interesting paths for future work.

First of all, scaling up for larger applications is an important point to convince the OCaml community to use FPGAs, and the FPGA community to use high-level languages. From a programmer's point of view, it would be useful to allocate values from the Macle code, support concurrent memory access, share Macle local functions (rather than inline them), and use more parallel skeletons, possibly, domain-specific.

Concerning the tool chain itself, we plan to switch to fully open source design and synthesis tools, with the idea that using such tools would facilitate both the static analysis of the Macle code and prediction of the efficiency of the generated hardware (e.g., resource usage and execution time). These informations could be used, for example, to decide which Macle function should be inlined and also to provide guarantees on applications interacting with the outside world. This is especially necessary for critical applications, for which it would be appropriate to use synchronous programming models with similarities to the HSML intermediate language used in the Macle compiler.

One can expect much higher speedups by using faster and/or more resourceful FPGA boards. Moreover, although the use of softcore processors leads to some inefficiencies, it remains very suitable for multi-core programming, each core carrying an instance of the OCaml VM.

In the longer term, we could also explore other ways to accelerate both the runtime (memory and exception management) and the VM interpreter by partially implementing them in hardware, or even using different levels of parallelism such as multiple VMs sharing Macle code. The latter could provide an interesting approach to exploit heterogeneous platforms including multi-cores, GPUs and FPGAs.

Acknowledgements Work on O2B and Macle is partially supported by the Center for Research and Innovation on Free Software (IRILL).

References

[1] R. Alur, S. Kannan, and M. Yannakakis, "Communicating hierarchical state machines," in *International Colloquium on Automata, Languages, and Programming*, Springer, 1999, pp. 169–178. DOI: 10.1007/3-540-48523-6_14.

[2] J. Auerbach, D. F. Bacon, P. Cheng, *et al.*, "Lime: a java-compatible and synthesizable language for heterogeneous architectures," in *ACM international conference on Object oriented programming systems languages and applications*, 2010, pp. 89–108. DOI: 10.1145/1869459.1869469.

[3] C. Baaij, M. Kooijman, J. Kuper, *et al.*, "Clash: structural descriptions of synchronous hardware using Haskell," in *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, IEEE, 2010, pp. 714–721. DOI: 10.1109/DSD.2010.21.

[4] J. Bachrach, H. Vo, B. Richards, *et al.*, "Chisel: constructing hardware in a Scala embedded language," in *DAC Design Automation Conference 2012*, IEEE, 2012, pp. 1212–1221. DOI: 10.1145/2228360.2228584.

[5] A. Canis, J. Choi, M. Aldham, *et al.*, "LegUp: high-level synthesis for FPGA-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays (FPGA)*, 2011, pp. 33–36. DOI: 10.1145/1950413.1950423.

[6] J. M. Cardoso, P. C. Diniz, and M. Weinhardt, "Compiling for reconfigurable computing: A survey," *ACM Computing Surveys (CSUR)*, vol. 42, no. 4, pp. 1–65, 2010. DOI: 10.1145/1749603.1749604.

[7] Y. Chi, L. Guo, J. Lau, *et al.*, "Extending high-level synthesis for task-parallel programs," in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, IEEE, 2021, pp. 204–213. DOI: 10.1145/3431920.3439470.

[8] J.-L. Colaço, G. Hamon, and M. Pouzet, "Mixing signals and modes in synchronous data-flow systems," in *Proceedings of the 6th ACM & IEEE International Conference on Embedded Software*, 2006, pp. 73–82. DOI: 10.1145/1176887.1176899.

[9] M. Danelutto, G. Mencagli, M. Torquati, *et al.*, "Algorithmic skeletons and parallel design patterns in mainstream parallel programming," *Int. J. Parallel Program.*, vol. 49, pp. 177–198, 2021. DOI: 10.1007/s10766-020-00684-w.

[10] J. Decaluwe, "MyHDL: a Python-based hardware description language," *Linux journal*, pp. 84–87, 2004.

[11] D. Drusinsky and D. Harel, "Using statecharts for hardware description and synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, no. 7, pp. 798–807, 1989. DOI: 10.1109/43.31537.

[12] J. Fumero, A. Stratikopoulos, and C. Kotselidis, "Running parallel bytecode interpreters on heterogeneous hardware," in *4th International Conference on Art, Science, and Engineering of Programming*, 2020, pp. 31–35. DOI: 10.1145/3397537.3397563.

[13] P. Gammie, "Synchronous digital circuits as functional programs," *ACM Computing Surveys (CSUR)*, vol. 46, no. 2, pp. 1–27, 2013. DOI: 10.1145/2543581.2543588.

[14] D. R. Ghica, A. Smith, and S. Singh, "Geometry of synthesis IV: compiling affine recursion into static hardware," in *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, 2011, pp. 221–233. DOI: 10.1145/2034574.2034805.

[15] S. Huang, K. Wu, H. Jeong, *et al.*, "Pylog: An algorithm-centric python-based FPGA programming and synthesis flow," *IEEE Transactions on Computers*, vol. 70, no. 12, pp. 2015–2028, 2021. DOI: 10.1109/TC.2021.3123465.

[16] Y. Ito and K. Nakano, "A hardware-software cooperative approach for the exhaustive verification of the Collatz conjecture," in *2009 IEEE International Symposium on Parallel and Distributed Processing with Applications*, IEEE, 2009, pp. 63–70. DOI: 10.1109/ISPA.2009.35.

[17] A. Kennedy, "Compiling with continuations, continued," in *12th ACM SIGPLAN International Conference on Functional programming*, 2007, pp. 177–190. DOI: 10.1145/1291151.1291179.

[18] Y.-H. Lai, E. Ustun, S. Xiang, *et al.*, "Programming and synthesis for software-defined FPGA acceleration: status and future prospects," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 14, no. 4, pp. 1–39, 2021. DOI: 10.1145/3469660.

[19] M. Maas, K. Asanović, and J. Kubiatowicz, "A hardware accelerator for tracing garbage collection," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2018, pp. 138–151. DOI: 10.1109/ISCA.2018.00022.

[20] A. Mycroft and R. Sharp, "A statically allocated parallel functional language," in *International Colloquium on Automata, Languages, and Programming*, Springer, 2000, pp. 37–48. DOI: 10.1007/3-540-45022-X_5.

[21] R. Nane, V.-M. Sima, C. Pilato, *et al.*, "A survey and evaluation of fpga high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, 2015. DOI: 10.1109/TCAD.2015.2513673.

[22] M. Papadimitriou, J. Fumero, A. Stratikopoulos, *et al.*, "Transparent compiler and runtime specializations for accelerating managed languages on FPGAs," *The Art, Science, and Engineering of Programming*, vol. 5, no. 2, pp. 8–1, 2020. DOI: 10.22152/programming-journal.org/2021/5/8.

[23] X. Saint-Mleux, M. Feeley, and J.-P. David, "SHard: a Scheme to hardware compiler," in *Workshop on Scheme and Functional Programming*, 2006.

[24] O. Segal, M. Margala, S. R. Chalamalasetti, *et al.*, "High level programming framework for FPGAs in the data center," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2014, pp. 1–4. DOI: 10.1109/FPL.2014.6927442.

[25] S. Singh and D. J. Greaves, "Kiwi: Synthesis of fpga circuits from parallel programs," in *16th International Symposium on Field-Programmable Custom Computing Machines*, IEEE, 2008, pp. 3–12. DOI: 10.1109/FCCM.2008.46.

[26] R. Stewart, K. Duncan, G. Michaelson, *et al.*, "RIPL: a parallel image processing language for FPGAs," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 11, no. 1, pp. 1–24, 2018. DOI: 10.1145/3180481.

[27] R. Townsend, M. A. Kim, and S. A. Edwards, "From functional programs to pipelined dataflow circuits," in *Proceedings of the 26th International Conference on Compiler Construction*, 2017, pp. 76–86. DOI: 10.1145/3033019.3033027.

[28] C.-J. Tsai, H.-W. Kuo, Z. Lin, *et al.*, "A Java processor IP design for embedded SoC," *ACM Transactions on Embedded Computing Systems*, vol. 14, no. 2, pp. 1–25, 2015. DOI: 10.1145/2629649.

[29] S. Varoumas, B. Vaugon, and E. Chailloux, "A generic virtual machine approach for programming microcontrollers: the OMicroB project," in *9th European Congress on Embedded Real Time Software and Systems (ERTS 2018)*, Jan. 2018.