Accelerating OCaml programs on FPGA

Loïc Sylvestre, Emmanuel Chailloux, Jocelyn Sérot

Noname manuscript No.
(will be inserted by the editor)

Accelerating OCaml programs on FPGA

Loïc Sylvestre · Emmanuel Chailloux · Jocelyn Sérot

Abstract This paper aims to exploit the massive parallelism of Field-Programmable Gate Arrays (FPGAs) by programming them in OCaml, a multi-paradigm, statically-typed language. It presents O2B, an FPGA-based implementation of the OCaml virtual machine using a softcore processor, running the entire OCaml language. It then introduces Macle, a language to express, in ML-style, hardware-accelerated user-defined functions. Macle exposes fine-grained parallelism available at the circuit level and enables to manipulate data structures dynamically allocated by OCaml programs. This hybrid approach, mixing Macle and OCaml codes, allows to easily prototype FPGA applications.

Keywords high-level programming, OCaml, virtual machine, FPGA, parallel computing, hardware acceleration, compiling, finite state machines

1 Introduction

Reconfigurable circuits, like Field-Programmable Gate Arrays (FPGAs), are suited to design custom architectures exploiting the concurrent nature of hardware structures [5]. The configuration of an FPGA is commonly produced by a synthesis toolchain supporting a hardware description language (HDL) such as VHDL or Verilog. Other examples of more expressive HDLs include Chisel [3] which is embedded in Scala, Clash [2] in Haskell, MyHDL [8] 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

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

Jocelyn Sérot
Institut Pascal, UMR 6602 UCA/CNRS/SIGMA
E-mail: jocelyn.serot@uca.fr

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

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 [19], FLOH [22] and Basic SCI [11], directly to RTL [10]. A representative example is SAFL (*Statically Allocated Parallel Functional Language*) [16], 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 [23] is a Java Virtual Machine (JVM) written in VHDL, calling a softcore processor[2] to handle dynamic class-loading. JikesRVM [15] is a JVM implemented on a CPU using an FPGA for accelerating automatic managing of dynamic memory (garbage collection / GC).
- High-Level Synthesis (HLS) promotes the use of imperative languages to design hardware [17]. Most of HLS tools, such as Catapult C or Handel-C, support a subset of C annotated with pragmas to optimize the compilation to RTL. LegUp [4] runs C programs on a softcore processor, or Pylog [12] on a hardcore processor, while compiling functions to RTL, (that do not use dynamic allocation and recursion).
- Other HLS tools[3] use OpenCL to express parallel applications and target heterogenous architectures involving Multicores, GPUs and FPGAs. TornadoVM [18], Aparapi [20] and GVM [9] implement the JVM in OpenCL. TAPA [6] is framework for task parallelism targeting OpenCL. These implementations, however, do not sufficiently expose the fine-grained parallelism available on the FPGA as well as their customization possibilities.
- FPGAs allows to implements parallel skeletons [7] and concurrency control constructs [6]. For instance, Lime [1] is a task-based data-flow programming language compiled to OpenCL or Verilog, and interacting with Java bytecode running on a CPU. Kiwi [21] is a subset of C♯ compiled to RTL and offering events, monitors and threads.

These approaches highlight several needs:

- runtime systems for high-level programming on FPGA using a softcore processor (like JAIP);
- partitioning between hardware accelerated code and a runtime (like Pylog);
- hardware acceleration of user-defined functions (like SAFL);
- parallel programming constructs (like Kiwi);
- uniformity between a host language and an embedded language used for acceleration (like Lime).

To fulfill these needs, we have ported on a softcore processor the OCaml VM and its runtime (including GC), to support the entire OCaml language. This VM approach is combined with hardware acceleration of functions expressed

[2] A Softcore processor is processor implemented in the reconfigurable part of an FPGA.
[3] Such as AMD Vivado HLS and Intel OpenCL SDK.

in an ML-like language extended with parallelism skeletons able to process data structures dynamically allocated by the OCaml runtime. This allows to take full advantage of the fine-grained parallelism of the FPGA, while programming it in a high-level way, in OCaml, allowing quick prototyping, static type-checking, simulation and debugging.

Our contributions are:

- O2B[4] (*OCaml On Board*), a port of the OMicroB [24] implementation of the OCaml Virtual Machine targeting the Nios II softcore processor implemented on an FPGA. O2B enables to call custom hardware accelerators from OCaml programs.
- Macle[5] (*ML accelerator*), a language to program, in ML-style, computation kernels to be accelerated (through a Macle to VHDL compiler). Such computation kernels, called *Macle circuits* thereafter, are used by the OCaml programs executed by O2B on FPGA. The interoperability layer between OCaml and the *Macle functions* is automatically generated. It includes C and OCaml code, VHDL descriptions and scripts to control the synthesis workflow. Macle offers language constructs to manipulate OCaml values, especially data structures (such as lists, arrays and matrices) allocated in the OCaml VM heap. In particular, 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. Section 2 introduces the O2B infrastructure to run OCaml programs on FPGA. Section 3 proposes a hybrid approach to accelerate OCaml programs augmented with Macle functions. Section 4 presents the compilation of Macle, using an intermediate language (HSML, *Hierarchical State Machine Language*) to abstract the VHDL target. Section 5 evaluates our approach on different benchmarks to measure the speedup resulting from using hardware-acceleration in Macle. Section 6 describes a mechanism using parallelism skeletons to optimize memory transfers when accessing the OCaml heap. Section 7 discusses the acceleration elements and programming style obtained and then identifies future work.

2 Customizable OCaml programs on FPGAs

O2B (*OCaml On Board*) is a tool to run OCaml programs on FPGAs. It is based on OMicroB [24], 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

Figure 1 describes the configuration process used to run OCaml programs on an Intel FPGA[6] via O2B. The OCaml 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 GC). The OCaml heap and stack are C static arrays. This program is associated with the functions of the Board Support Package (BSP lib) 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.

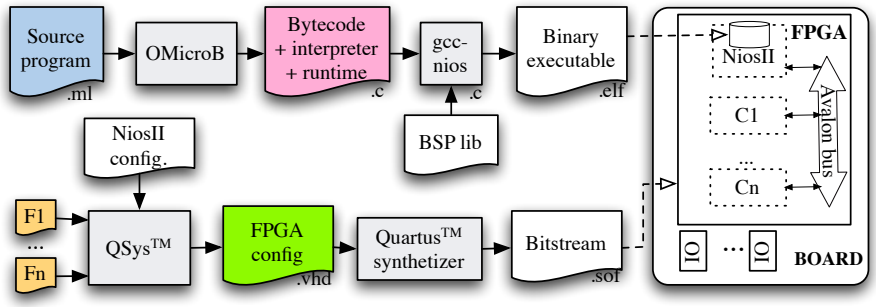


Fig. 1 Compilation flow targeting Intel FPGAs

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 external DRAM. 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 manually mapped into the memory of the softcore processor.

Figure 2 shows the source code of an OCaml program designed to run with O2B. It defines three implementations of the `gcd` (*the greatest common divisor*) algorithm. The difference of 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 microsecond.

OCaml code

```
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 ();;
```

C code

```
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);
}
```

Fig. 2 An OCaml program executable by O2B

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 arguments (resp. retrieving the result) to (resp. from) the corresponding dedicated registers of the custom component. In figure 2, the corresponding operations are abstracted by the macros `GCD_ARG`, `GCD_START`, `GCD_RDY` and `GCD_RESULT`).

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

Moreover, describing the behavior of the component, in synthetizable VHDL, is tedious. For the GCD example, describing this behavior and exchanging the arguments and result respectively requires 50 and 100 lines of VHDL. 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. With the compilation flow introduced in the next section, RTL descriptions of custom components as well as glue code between OCaml and these components (including OCaml, C and VHDL files) will be automatically generated from a high-level formulation in the Macle language.

3 A hybrid approach for high-level FPGA programming

The O2B experiment described in the previous section enables to run OCaml programs on FPGA via a softcore processor and call hardware accelerators from them. The difficulty is still to program these accelerators and synthesize them on the same FPGA as the softcore. In this section, we propose to express these accelerators in an ML-like language compiled to RTL. This language, called Macle (*ML Accelerator*), can inter-operate with the OCaml runtime of O2B and therefore can be used to accelerate OCaml *host* programs on FPGA.

3.1 Compilation Flow

Figure 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. OCaml code is compiled to bytecode to be executed by O2B targeting a softcore processor implemented on the FPGA.

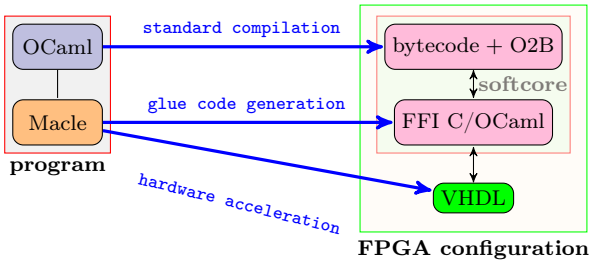


Fig. 3 An hybrid approach to run OCaml programs on FPGA via O2B and Macle

Each Macle circuit is a function compiled to VHDL and then synthesized as a custom hardware component usable from OCaml programs. The glue code is generated from the inferred type of the Macle circuit. The FPGA configuration

is automatic and easily programmable without prior knowledge of hardware description languages.

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.

Figure 4 defines the syntax of Macle.

The left side of the figure defines 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 $\mathcal{X}$), constants, application 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 call uses an implicit parallel let-binding to compute the arguments passed to the function. Non-recursive functions can take functions as arguments[9].

Macle Core		**Interaction with OCaml**	
circuit	$ci ::=$ **circuit** $f\ \overrightarrow{x} = e$	exception	$exn ::=$ `Failure` $\langle string \rangle$
constant	$c ::=$ **true** | **false** | $\langle integer \rangle$ | `()`	pattern	$p ::= C \mid C(x_1, \cdots x_n)$
variable	$x, y, f \in \mathcal{X}$	expression	$e ::= \ldots$
operator_1	$\ominus ::= - \mid$ **not** $\mid \ldots$		| **raise** exn
operator_2	$\oplus ::= + \mid \ < \ \mid \ldots$		| **match** e **with** $\overrightarrow{p \to e'}$
expression	$e ::= x \mid c \mid \ominus e \mid e_1 \oplus e_2$		| ! e
	| **if** e **then** e_1 **else** e_2		| $e := e'$
	| **let** $x_1 = e_1$ **and**		| $e.(e')$
	$\cdots \ x_n = e_n$ **in** e'		| $e.(e') \leftarrow e''$
	| **let** $f\overrightarrow{x} = e$ **in** e		| **array_length** e
	| **let rec** $\overrightarrow{f\overrightarrow{x} = e}$ **in** e'		| $e \ ; \ e'$
	| $f\ \overrightarrow{e}$		| **for** $x = e$ **to** e' **do** e''
	| $\ldots$		**done**

Fig. 4 Syntax of the Macle language

The right side of Figure 4 presents the Macle constructs used to interact with the OCaml runtime :

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

- `!`e for accessing to the content of the reference e;
- $e \coloneqq e'$ for setting the content of the reference e to the value of e';
- e`.(`e'`)` for accessing to 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.
- for raising a built-in exception parametrized by literal strings,
- for surface pattern matching on algebraic datatypes (ADT),

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

Finally, the sequence e ; e' is a syntactic sugar for **let** $x = e$ **in** e' where x is a fresh name. *For-loops* are encoded with *let-rec*.

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.

Figure 5 shows three Macle circuits and an OCaml program calling a Macle circuit. The circuit `gcd_rtl` expresses the *Gcd* algorithm in Macle Core. The circuit `rev` reverses the order of the elements of an OCaml array. The circuit `collatz` computes the *stopping time* of a Collatz [13] sequence (also called *Syracuse*) starting from a given integer.

Computations in Macle

```
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 rev a =
  let n = array_length a in
  for i = 0 to (n-1) / 2 do
    let t = a.(i) in
    a.(i) <- a.(n-1-i);
    a.(n-1-i) <- t
  done ;;
```

Mixing OCaml and Macle codes

```
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();;
```

Fig. 5 Examples of Macle circuits and call from OCaml program

The circuit `eval_exp` evaluates an abstract syntax tree allocated in the OCaml heap. It safely accesses the OCaml heap 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** $\cdots$ **with** construct. This program evaluates the expression `Add(Int(1),Var(0))` recursively and prints the result. Evaluate `Var(0)` fetches the value at the index 0

of the array `env = [|100|]`. Recursion in Macle uses an explicit call stack, as described in section 5. Tail-recursion does not require a stack.

4 Compiling Macle

The global compilation flow from Macle to VHDL is depicted Figure 6. It involves four passes. The first pass 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* [14] (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 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 flattens the hierarchical structure of HSML. The fourth pass translates a flat HSML description into VHDL.

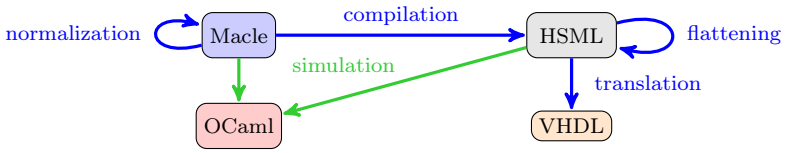


Fig. 6 Compilation flow of Macle to VHDL

At each point of the compilation flow, an OCaml backend is provided for simulation and debugging on a PC.

Due to space limitations, the rest of this section only describes the compilation of Macle Core to HSML.

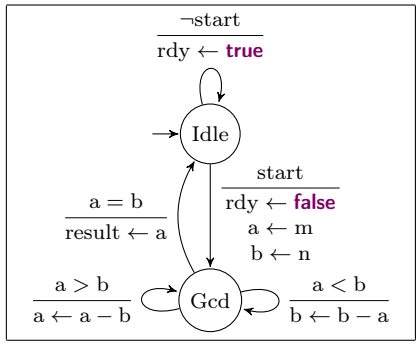
4.1 Targeting the register transfer level

Synchronous finite state machines (FSM) are commonly used to describe computations at the register transfer level (RTL). A 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.

Figure 7 gives a graphical representation of a FSM describing the computation of a `gcd` function and its encoding in VHDL.




```
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 Figure 5)

The `start` input and `rdy` output are used respectively to start and signal the end of the computation. In the VHDL code, modifications of the state variable `STATE` as well as the outputs and local variables are denoted using signal assignments (`<signal_name> <= <expression>`). Assignments performed 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. Note that in the code given Figure 7, arguments and result

are encoded as 31-bit signed integers. This is to have the same representation of OCaml value than in the O2B runtime, in order to call this circuit from OCaml programs.

By declaring separate processes, each encoding a given FSM, within the same entity/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 HSML : a 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.

Figure 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. A FSM is a set of 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 a FSM A. HSML offers a notion of hierarchy. For instance, a FSM **let rec** t_1 **and** $\cdots$ t_m **in** (**let rec** t'_1 **and** $\cdots$ t'_n **in** $f()$) is a hierarchical formulation of the FSM **let rec** t_1 **and** $\cdots$ t_m **and** t'_1 **and** $\cdots$ t'_n **in** $f()$.

circuit	$\phi ::=$ **circuit** f $\overrightarrow{x_{in}}$ **returns** $\overrightarrow{x_{out}}$ $=$ **var** $\overrightarrow{x}$ **in** P
parallel composition	$P ::= A_1 \parallel \cdots A_n$
FSM	$A ::=$ **let rec** ts **in** A_{init} $\mid$ **if** e **then** A_1 **else** A_2 $\mid$ **do** $x_1 \leftarrow e_1$ **and** $\cdots$ $x_n \leftarrow e_n$ **then** A $\mid f()$ $\mid P$ **in** A
transitions	$ts ::= \epsilon \mid f_1() = A_1$ **and** $\cdots$ $f_n() = A_n$
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

A HSML expression e is a variable, a constant or the application of a built-in operator. The construct (**do** $x_1 \leftarrow e_1$ **and** $\cdots$ $x_n \leftarrow e_n$ **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.

Figure 9 shows an HSML circuit corresponding to the VHDL code given Figure 7. This circuit was automatically generated from the Macle circuit `gcd_rtl` defined Figure 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 RT level (described informally on the VHDL code of Figure 7) while offering a notion of hierarchy which makes it close to an expression language. In particular, some HSML constructs (like *let rec* and conditional) are common with Macle. Thus, HSML constitutes a useful intermediate language for compiling Macle to VHDL.

4.3 Compiling Macle Core

The compilation $\mathcal{C}[\![\textbf{circuit } f\,\overrightarrow{x} = e]\!]$ 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.

$$\mathcal{C}_{\mathsf{ci}}[\![\textbf{circuit } f\ \overrightarrow{x} = e]\!] = \textbf{circuit } f\ \overrightarrow{x_{in}} \textbf{ returns } \overrightarrow{x_{out}} = \textbf{var } \overrightarrow{x_{local}} \textbf{ in } \overbrace{\mathcal{C}[\![e]\!]^{start,rdy,result}}^{s}$$
$$\text{where } \begin{cases} \overrightarrow{x_{in}}, \overrightarrow{x_{out}} \text{ and } \overrightarrow{x_{local}} \text{ are inputs, outputs and local} \\ \text{variables declarations inferred from } s \\ start, rdy, result \text{ are fresh names} \end{cases}$$

The compilation $\mathcal{C}[\![e]\!]^{start,rdy,result}$ 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 $\mathcal{C}_{\mathsf{e}}[\![e]\!]_{\rho}^{result,idle}$ is defined next. The compilation environment ρ maps functions names to the list of their formal arguments.

$$\begin{aligned} \mathcal{C}[\![e]\!]^{start,rdy,result} = \ & \textbf{let rec } idle() = \\ & \quad \textbf{if } start \textbf{ then } (\textbf{do } rdy \leftarrow \textbf{false then } \mathcal{C}_{\mathsf{e}}[\![e]\!]_{\emptyset}^{r,idle}) \\ & \quad \textbf{else } (\textbf{do } rdy \leftarrow \textbf{true then } idle()) \\ & \textbf{in } (\textbf{do } rdy \leftarrow \textbf{true then } idle()) \\ & \text{where } idle \text{ is a fresh name} \end{aligned}$$

The compilation $\mathcal{C}_{\mathsf{e}}[\![e]\!]_{\rho}^{result,idle}$ of a subexpression is inductively defined on the syntax of the expressions. The compilation of a subexpression e which do not contain control structures is defined as an affectation of e to a variable *result*

continuing with a tail-call to a destination.

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

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

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

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 $\rho[f/(x_1,\ \cdots\ x_n)]$, 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 $f()$.

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

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

The compilation $\mathcal{C}_{\mathsf{e}}[\![\textsf{let } x = e \textsf{ in } e']\!]_{\rho}^{r,idle}$ 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'.

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

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*.

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

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

Since they expose parallelism, 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, following our hybrid approach. These

programs are compared by taking as reference equivalent C code running on the same softcore processor. We first consider programs written in Macle Core (as described on the left side of figure 4), and then Macle circuits interacting with the OCaml runtime (right side of figure 4).

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); 1,638 Kb of on-chip memory; a clock frequency of 50 MHz[10]. 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 both implemented with C static arrays, both stored in the on-chip memory. The whole is compiled via the Nios II backend of `gcc` with optimizations enabled (-Os). All data structures manipulated by OCaml, C and Macle code using the OCaml heap and the OCaml arrays bounds are dynamically checked at each access.

Measuring elapsed time on a FPGA Macle circuits are called from a C block running on the softcore. Indeed, as described in section 2.2, is necessary to write arguments in the dedicated registers of the custom component implementing the circuit, start the circuit and wait for the end of the computation to read the result (again in the dedicated registers of the custom component). We measure the execution time of each Macle circuit from the beginning to the end of the corresponding C block.

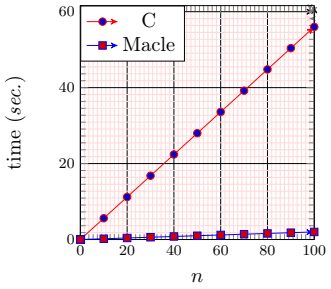
5.2 Macle Core

We here assess the efficiency gains obtained both by rewriting a C function as a Macle circuit and, possibly, replicating this circuit to parallelize the corresponding computations.

Pure Computations Figure 10 shows the execution time of the `gcd_rtl` Macle circuit (given Figure 5) and the `gcd_c` C function (given Figure 2) called by an OCaml program. The observed Macle *vs* C speedup factor is 30.

A similar experiment with the Macle circuit `collatz` (given Figure 5) leads to a $\times 60$ speedup. The hardware implementation of `gcd_rtl` and `collatz` both use approximately 360 logic elements (LEs), *i.e.* 0.75% of the total available on the target FPGA used here.

[10] The DE10-Lite is also equiped with a 64 Mb external SDRAM but it is not used in this series of experiments.




```
(* test program *)
let main () =
   let n = 40 in
   let nb_it = n * (1000 * 1000) in

   print_int (gcd_c nb_it 1);
   print_int (gcd_rtl nb_it 1) ;;

main () ;;
```

Fig. 10 Execution time of a simple computation (gcd) in Macle and C

Parallel computations Figure 11 gives a circuit `sum_gcd2` calling twice a function `gcd_rtl` and combining results. The **let** ··· **and** ··· **in** ··· constructs is implemented by a synchronization barrier involving a parallel composition of two instances of the FSMs given Figure 7.

```
circuit sum_gcd2 a1 ··· an 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 ···
  and xn = gcd an y in
  (x1 + ··· xn)
```

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

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

The global execution time of the barrier is the max of the execution times of the expressions (`gcd` $\texttt{a}_i$ `y`), to which is added the execution time of the rest of the computation (here instantaneous). For instance, calling the circuit $\texttt{sum\_gcd}_2$ with equal arguments a_1 and a_2 doubles the previous $\times 30$ speedup observed in Macle *vs* C (Figure 10). Generalizing this example to circuits $\texttt{sum\_gcd}_n$ (computing n times `gcd_rtl` and summing results) gives a speedup of $30 \times n$ in Macle *vs* C (e.g., $\texttt{sum\_gcd}_{32}$ is 960 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 on the right side of Figure 11.

5.3 Interacting with the OCaml runtime

Macle enables hardware acceleration for the functional-imperative fragment of OCaml, accessing directly the OCaml heap (a shared memory allocated in the RAM) using a bus.

Left side of Figure 12 shows the execution time of a Macle circuit `product` multiplying two integer matrices of size $n \times n$, *vs* a C version. The Macle version is 27 times faster than the C one. The generated hardware uses 1602 LEs.

Right side of Figure 12 shows the execution time of the Macle circuit `eval_exp` (given Figure 5) *vs* an OCaml version, recursively evaluating trees of arithmetic expressions of various sizes (in number of constants and variables). Note that the realization of this Macle circuit uses 17,566 LEs because it requires an explicit stack which is (here) implemented using LEs instead of on-chip memory blocks. The resulting speedup is encouraging: the Macle circuit (using a recursive formulation) is 23 times faster than the C formulation.

From a programmer's point of view, this speedup is simply obtained by replacing a **let** keyword in the original OCaml formulation by "**circuit**".

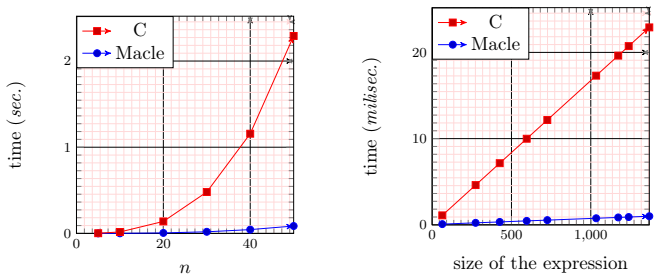


Fig. 12 Execution time of Macle circuits using imperative features (a) and recursion (b)

This preliminary evaluation shows that reformulating side-effect-free C functions as Macle circuits can bring substantial speedups (eg., up to 30 for the `gcd_rtl` of Figure 5). Replicating the hardware corresponding to these circuits, intrinsically resulting in their parallel execution, allows to further boosts these speedups (e.g., up to 960 for the $\texttt{sum\_gcd}_{32}$ example given Figure 10).

Macle also offers computations on data structures dynamically allocated in the VM heap and accessed in an imperative manner. But for large data structures, such as arrays, the cost of accessing the corresponding memory can quickly create a bottleneck, as discussed in the next section.

6 Optimised tranfers and parallelism skeletons

Allowing Macle circuits to manipulate values stored in the OCaml heap has a cost. Because this heap is implemented in shared memory[11], each access requires a bus transaction. When manipulating large data structures, like arrays, the corresponding overhead can quickly become prohitive. To overcome this problem, Macle provides some dedicated constructs, called *parallelism skeletons* aiming at minimizing this overhead and offering higher-level parallelism. These skeletons are listed Figure 13.

[11] On-chip memory in our experimental platform, but the problem would be worst if the heap was allocated in external DRAM.

$\textsf{array\_map}\langle n\rangle : (\alpha \to \beta) \to \alpha\ \texttt{array} \to \beta\ \texttt{array} \to \texttt{unit}$
$\textsf{array\_reduce}\langle n\rangle : (\alpha \to \beta \to \alpha) \to \alpha \to \beta\ \texttt{array} \to \alpha$
$\textsf{array\_scan}\langle n\rangle : (\alpha \to \beta \to \alpha) \to \alpha \to \beta\ \texttt{array} \to \alpha\ \texttt{array} \to \texttt{unit}$

Fig. 13 Simple parallelism skeletons available in Macle

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

For instance, the expression ($\textsf{array\_map}\langle 64\rangle$ f `src dst`) copies the 64 first elements of the OCaml array `src` into a VHDL array, computes the function f *in parallel* on each element of this array and writes back the 64 resulting values in the OCaml array `dst`. Processing the whole OCaml array is carried out by iterating this transfer-execution-transfer sequence.

Figure 14 is a simple OCaml program mixing imperative features, computations and a parallelism skeleton $\textsf{array\_map}\langle k\rangle$ within a Macle circuit $\texttt{filter\_mul}_k$. It implements the Eratosthene sieve: determining all the prime numbers less than a natural number n, by filtering an OCaml array of size n containing integer from 1 to n. The circuit $\texttt{filter\_mul}_k$ removes array elements that are multiple of a given integer y using the `gcd` algorithm. This computation is performed in parallel by group of k elements of the array, encoding the removed elements by the integer zero. The current prime number used to filter the rest of the array is determined by a loop traversing the array, element by element, skipping zeros (i.e., elements already removed).

Macle code

```
circuit filter_mul_k 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
  let remove x =
    if x <= 1 then 0 else
    if x == y then x else
    if gcd x y == 1 then x else 0
  in

  if y <= 1 then () else

  array_map<k> remove a a ;;
```

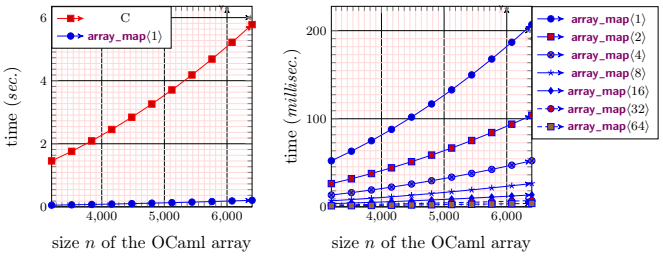
OCaml code

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

let print_if_not_zero x =
  if x != 0 then print_int x ;;

let eratostene_k a =
  for i = 1 to Array.length a - 1 do
    filter_mul_k src.(i) a
  done ;;

let main() =
  let n = (32*100) in
  let a = interval n in
  eratostene_k a;
  Array.iter print_if_not_zero a ;;

main();;
```

Fig. 14 A Macle circuit with a parallelism skeleton computing the Eratosthene sieve

Figure 15 shows, according to k, the size (in LEs) of the $\texttt{filter\_mul}_k$ circuit and the execution time of $\texttt{filter\_mul}_k$ with argument y being 2 and a being `(interval` n`)`. Results are compared to a sequential C version. Doubling the degree of parallelism k almost doubles both the size of the circuit and the

speedup (taking into account the transfer time). For instance, `filter_mul`$_{64}$ is 53 times faster than `filter_mul`$_1$. Moreover, `filter_mul`$_1$ is 28 times faster than the C version, resulting in a cumulated speedup of $53 \times 28 = 1{,}484$.



k	**size (LEs)**
1	1,240
2	1,655
4	2,487
8	4,182
16	7,521
32	15,107
64	29,739

Fig. 15 space/time trade-off of the Macle circuits `filter_mul`$_k$ and comparison with C

7 Conclusion

In this paper, we proposed an hybrid 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.

Macle, as well as the intermediate language HSML used by the Macle compiler, are statically typed and this feature provides much stronger guarantees on the safety of the generated circuits than using classical HDLs.

We described an implementation of this approach based on the O2B platform and a complete compilation flow from Macle circuit descriptions to VHDL. This compilation flow is fully automatized and easy to use. Moreover, it includes a simulation mode generating OCaml code from different points of the compiler to test the applications on PC before loading them on FPGA.

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 hosted softcore) can be obtained by combining the ability to compile a function to hardware and

the possibility to replicate the corresponding hardware in order to use data parallelism. Parametrizable parallelism skeletons both offer a way to tackle the bottleneck occurring when exchanging data between the OCaml program and the accelerated functions and also a very practical way to explore the time *vs.* space trade-off, a classical issue when programming FPGAs (reducing computing time by increasing the number of used logic elements).

The work described here offers many interesting paths for future work.

First of all, scaling up for larger applications, both symbolic and numerical, is an important point to convince the OCaml community to use FPGAs, but also the FPGA community to use high level languages. For this, a technical but critical issue is the ability to use larger, external memory chips, with optimized transfers (using DMA facilities for example) to store large dynamically allocated data structures. The ability to implement local stacks used by circuits to realize non-tail recursion (such as evoked in section 5.3) in on-chip memory (instead of LEs) is another key point to allow large and complex symbolic computations to be implemented on moderately-sized FPGAs. From a programmer's point of view, the definition and implementation of new parallelism skeletons, including, possibly, domain-specific skeletons, could also help.

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 the static analysis of the Macle circuits and the prediction of the space and time characteristics of the generated hardware (LE usage and execution time). These information could be used, for example, to decide which circuit should be duplicated, and also to provide guarantees on applications interacting with the outside world, including critical applications using synchronous programming models (close to synchronous FSMs).

In a 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 as circuits, or even try to create applications using different levels of parallelism by using multiple VMs sharing Macle circuits. The latter could provide an interesting approach to exploit heterogeneous platforms including multi-cores, GPUs and FPGAs for example.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.
[6] 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.
[7] 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.
[8] J. Decaluwe, "MyHDL: a Python-Based Hardware Description Language," *Linux journal*, pp. 84–87, 2004.
[9] 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.
[10] P. Gammie, "Synchronous digital circuits as functional programs," *ACM Computing Surveys (CSUR)*, vol. 46, no. 2, pp. 1–27, 2013.
[11] 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.
[12] 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.
[13] 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.
[14] A. Kennedy, "Compiling with continuations, continued," in *12th ACM SIGPLAN International Conference on Functional programming*, 2007, pp. 177–190.
[15] 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.
[16] A. Mycroft and R. Sharp, "A Statically Allocated Parallel Functional Language," in *International Colloquium on Automata, Languages, and Programming*, Springer, 2000, pp. 37–48.
[17] 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.
[18] 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.
[19] X. Saint-Mleux, M. Feeley, and J.-P. David, "SHard: a Scheme to Hardware Compiler," in *Workshop on Scheme and Functional Programming*, 2006.
[20] 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.
[21] 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.
[22] 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.
[23] 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.
[24] 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.