Programming parallelism on FPGAs with Eclat

Loïc Sylvestre, Jocelyn Sérot, Emmanuel Chailloux

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

Programming parallelism on FPGAs with Eclat

Loïc Sylvestre · Jocelyn Sérot · Emmanuel Chailloux*

Abstract Eclat is a general purpose OCaml-like programming language with synchronous semantics for designing reactive hardware applications on FPGAs. It is compiled down to hardware descriptions to realize computations as intrinsically parallel circuits able to interact with the physical world.

This paper presents a formalization of Eclat with shared memory and mutability, handling concurrent memory accesses while preserving determinacy. The language is precise enough to express efficient circuits with fine control over throughput, parallelism and time-space trade-off. It also enables abstraction by letting the programmer implement, reuse and compose algorithmic skeletons such as *map* and *pipe*. The synchronous approach makes it possible to estimate the execution time of parallel programs by simple reasoning on source code for quick prototyping and optimization.

Keywords Language design and implementation, FPGAs,
Performance prediction, Algorithmic skeletons, Synchronous programming

1 Introduction

Field-Programmable Gate Arrays (FPGAs) are digital circuits interconnecting generic logic cells, memory blocks and I/O blocks; all of these elements are configurable by logic synthesis to implement custom circuits. This architecture offers an interesting trade-off between Application-Specific Integrated Circuits (ASICs) and general-purpose processors. It exposes fine-grained parallelism

* This Work is partially supported by the Center for Research and Innovation on Free Software (IRILL).

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

with very low latency in I/Os processing. This constitutes a valuable target for both embedded system design [5] and heterogenous computing.

FPGAs are classically programmed as synchronous circuits at the so-called Register Transfer Level (RTL) in Hardware Description Languages (HDLs) such as VHDL or Verilog. The main advantage of RTL is predictability because any architectural detail (such as throughput, latency and parallelism) must be encoded in terms of register updates at each clock tick of a global clock.

A variety of *high-level synthesis* tools have been proposed to generate hardware descriptions from software-like (C/OpenCL) code [12]. However, one of the side effects of such automation is that many architectural details are hidden from the programmer [7]. This is a source of unpredictability, *e.g.*, for timing.

To trade this issue, we are currently developing the *high-level* programming language Eclat (*declarative language*), which is tailored for the design of real-time hardware applications on FPGAs [16].

Eclat is an OCaml-like language compiled to VHDL and whose formal semantics is based on the synchronous model of computation [4]. It reflects, at the language level, the tick of the physical clock driving the target FPGA: program execution is discretized as a sequence of computation steps called *clock cycles* (or *instants*). The duration of each cycle is fixed by the FPGA clock.

Eclat programs can mix several programming styles, such as high-level hardware description, general purpose computation and interaction with the physical world. This paper focuses on general-purpose parallel computing on FPGAs with Eclat. The contributions are:

- the design of Eclat, including an operational semantics to model shared memory, mutability and concurrency in a deterministic manner;
- the implementation[1] of these new features in the Eclat compiler[2];
- the exploration of space-time trade-off in Eclat, by mixing fine-grained parallelism and shared memory, to obtain important gains of speed on an FPGA with regard to sequential implementations running on a CPU;
- the efficient implementation of algorithmic skeletons using Eclat with performance prediction and optimization by simple reasoning on source code.

Section 2 formalizes the Eclat language core and mentions practical extensions that are used in the rest of the paper. Section 3 presents the Eclat support for array allocation in shared memory and mutability; the mix of synchrony and shared memory provides, for free, determinacy and fairness for concurrent memory accesses, which constitues a basis for performance prediction. Follows an evaluation, in Section 4, which shows how to exploit fine-grained parallelism with Eclat, while using shared memory, in order to achieve high performance on quite large inputs. Section 5 considers the parallel implementation of algorithmic skeletons (as higher-order functions operating on Eclat arrays); it focusses on composability and performance prediction through a

[1] The artefact presented in this paper, including the source code for the benchmarks, is available online: https://github.com/lsylvestre/HLPP24

[2] We mainly target Intel FPGAs, but also provide experimental backends for Xilinx FPGAs and the open-source synthesis tool-chain Yosys.

simple example, paving the way for understanding and optimizing more complex skeleton-based Eclat applications. Section 6 discusses related work, and section 7 highlights future research perspectives.

2 The Eclat language

This section presents[3] Eclat, a declarative programming language compiled to synchronous circuits for reconfiguring FPGAs. The FPGA target is crucial, here, because it exposes a physical clock that the language exploits to finely control timing, throughput and parallelism in the applications.

Eclat is based on a core language named λ_{cbv}, which is a typed left-to-right call-by-value λ-calculus with pairs and like immutable vectors, parametric polymorphism and a limited form of higher-orderness to be efficiently compiled to a synchronous circuit.

Eclat also features tail-recursion and coarse-grained parallelism, both relying on the concept of *pause* [9]. A pause is a global synchronization barrier used in Esterel-like synchronous languages to model time *logically* as a discrete sequence of clock cycles. In Eclat, each recursive function call performs an implicit pause which delays the execution of the function body to the next clock cycle. Following the path initiated by Esterel [6], the hardware implementation of Eclat directly connects logical time to the clock of the FPGA.

2.1 Syntax and dynamic semantics of λ_{cbv}

The syntax of λ_{cbv} is shown in the upper part of Fig. 1. We let f, g, x, y and z range over an infinite set of variables. Each program π is an expression e, which computes an immediate value $\mathring{v}$ of basic type $\mathring{\tau}$, *i.e.*, a constant or a pair of immediate values. Among constants, immutable vectors (*resp.* integers) have a size s in number of elements (*resp.* in number of bits). Integers are signed. Values are constants, pairs, and functions. Local definition (*let*) and function abstraction (*fun*) deeply destructurate pairs of values matching a pattern p. Conditional (*if*) evaluates only one branch according to a condition.

Following a Hindley-Milner type discipline, a type scheme σ associated with a let-binding is a type where some variables (α for types, $\mathring{\alpha}$ for basic types and η for sizes) are quantified to be instantiated (by types, basic types or sizes respectively) when using this binding. All functions (including operators) are unary[4]; this is ensured at type level by enforcing functions to return an immediate value $\mathring{v}$ of basic type $\mathring{\tau}$. Operators over vectors are:

- `vect_create<`n`>` : $\forall\mathring{\alpha}\cdot\ \mathring{\alpha} \to \mathring{\alpha}$ `vect<`n`>` for creation with an initial value, *e.g.*, `vect_create<`n`>(`c`)` evaluates to the array `{`$c, \cdots\ c$`}` of size n,
- `vect_size` : $\forall\eta\cdot\forall\mathring{\alpha}\cdot\ \mathring{\alpha}$ `vect<`η`>` $\to$ `int` for access to the size of the vector,

[3] This presentation focuses on parallel programming in Eclat and therefore omits two Eclat constructs (named *register* and *exec*) [16], which are used for interacting with the physical I/Os in a (Lustre/Scade like) synchronous dataflow programming style.

[4] For instance, the operator `(+)` has type $\forall\eta\cdot$ `(int<`η`>` $\times$ `int<`η`>)` $\to$ `int<`η`>`.

- `vect_nth` : $\forall\eta\cdot\forall\mathring{\alpha}\cdot$ `(`$\mathring{\alpha}$ `vect<`η`> ×` $\mathring{\alpha}$ `int)` $\to \mathring{\alpha}$ for access to the element at the given computed index modulo the size,
- `vect_copy_with` : $\forall\eta\cdot\forall\mathring{\alpha}\cdot$ `(`$\mathring{\alpha}$ `vect<`η`> ×` `int` $\times\ \mathring{\alpha}$`)` $\to \mathring{\alpha}$ `vect<`η`>` for copy with update of the element at the given computed index,
- `vect_mapi` : $\forall\eta\cdot\forall\mathring{\alpha}\cdot\forall\mathring{\beta}\cdot$ `(((int` $\times\ \mathring{\alpha}$`)` $\to \mathring{\beta}$`)` $\times\ \mathring{\alpha}$ `vect<`η`>)` $\to \mathring{\beta}$ `vect<`η`>` transforming a vector by mapping a function on indexes and elements.

In concret syntax, programs can have global declarations "**let** $x_1 = e_1$`;;` $\cdots$ **let** $x_n = e_n$`;;`". Expressions can be over-parenthesized, *e.g.*, `(`f`(`g`(`x`)))`. Tuples are provided as generalization of pairs. Size annotations for integer literals can be omitted (*i.e.*, expression n stands for the constant `(`n`:int<`s`>)` with s a fresh size variable). We let type `int` be an alias for `int<32>` and we use infixe notation for binary operators as well as standard OCaml-like notations, *e.g.*, **let** f `(`p`:`τ`):`τ' `=` e`;;` stands for **let** f `=` `((`**fun** $p \to e$`):`$\tau \to \tau'$`);;`.

program	π ::= e	**operator**	op ::= `&` | `or` | `xor` | `not` | `+` | `-` | `*` | `/` | `mod` | `<` | `>` | `<=` | `>=` | `=` | `<>`
expression	e ::=	*vector creation*	| `vect_create<`n`>`
variable	x	*vector access*	| `vect_size` | `vect_nth`
constant	| c	*vector update*	| `vect_copy_with`
pair	| `(`e`,`e`)`	*transformation*	| `vect_mapi`
local def.	| **let** $p = e$ **in** e	**immediate value**	$\mathring{v}$ ::= c | `(`$\mathring{v}$`,`$\mathring{v}$`)`
application	| $e\ e$	**value**	v ::= $\mathring{v}$ | `(`v`,`v`)` | **fun** $p \to e$ | op
abstraction	| **fun** $p \to e$	**basic type**	$\mathring{\tau}$ ::= `unit` | `bool` | `int<`s`>` | $\mathring{\tau} \times \mathring{\tau}$ | $\mathring{\tau}$ `vect<`s`>` | $\mathring{\alpha}$
operator	| op	**type**	τ ::= $\mathring{\tau}$ | $\tau \times \tau$ | $\tau \to \mathring{\tau}$ | α
type constr.	| `(`e `:` τ`)`	**size**	s ::= n | η
conditional	| **if** e **then** e **else** e	**type scheme**	σ ::= $\forall\alpha\cdot\sigma$ | $\forall\mathring{\tau}\cdot\sigma$ | $\forall\eta\cdot\sigma$ | τ
pattern	p ::= x | `()` | `(`p`,`p`)`		
constant	c ::= `true` | `false` | `()`		
sized int.	| `(`n `:` `int<`s`>)`		
vector	| `{`c_1`,` $\cdots$ c_n`}`		

evaluation context F ::= `(`$\square$`,`e`)` \| `(`v`,`$\square$`)` \| **let** $p = \square$ **in** e \| $\square\ e$ \| $v\ \square$ \| **if** $\square$ **then** e **else** e

$$\textsc{Context}\quad \frac{e/\mu \longrightarrow e'/\mu'}{F[e]/\mu \longrightarrow F[e']/\mu'}$$

$$\textsc{Ty-constr}\quad (e{:}\tau)/\mu \longrightarrow e/\mu$$

$$\textsc{Let}\quad \textbf{let}\ p = v\ \textbf{in}\ e/\mu \longrightarrow e[p \mapsto v]/\mu$$
$$\textsc{Fun-app}\quad (\textbf{fun}\ p \to e)\ v/\mu \longrightarrow e[p \mapsto v]/\mu$$
$$\textsc{Op-app}\quad op\ v/\mu \longrightarrow \vartheta(op, v, \mu)$$
$$\textsc{If-true}\quad \textbf{if}\ \texttt{true}\ \textbf{then}\ e_1\ \textbf{else}\ e_2/\mu \longrightarrow e_1/\mu$$
$$\textsc{If-false}\quad \textbf{if}\ \texttt{false}\ \textbf{then}\ e_1\ \textbf{else}\ e_2/\mu \longrightarrow e_2/\mu$$

Fig. 1 Syntax and reduction semantics of λ_{cbv}

The lower part of Fig. 1 defines the operational semantics of λ_{cbv} as a reduction semantics in the style of Felleisen [8]. The reduction relation $e/\mu \longrightarrow e'/\mu'$ rewrites configurations of the form e/μ, where an expression e is paired with a memory μ. Memory is kept abstract in this section (it will be defined in section 3 when adding mutable arrays in shared memory).

An evaluation context F is an expression with a hole $\square$. $F[e]$ is the expression obtained by substituting the hole $\square$ by expression e in context F. The context `(`v`,`$\square$`)` specifies the evaluation order: the right component of a pair cannot be reduced until the left component is a value. Rule Context reduces

the expression e in the hole of any context $F[e]$ and propagates memory modifications (if any). Other rules specify, in a deterministic way, the behavior of each language feature. For example, reducing a let-binding or a function application performs a substitution without capture $e[p \mapsto v]$ of pattern p by value v in expression e (rules Let and Fun-app). Operator application is defined by semantic function ϑ (rule Op-app), *e.g.*, $\vartheta(\texttt{not}, \texttt{true}, \mu) = \texttt{false}/\mu$.

An important property of λ_{cbv} is termination: for any well-typed configuration e/μ, it exists a finite sequence of zero or more reductions $e/\mu \rightarrow^{\star} e_n/\mu_n$ such that e_n is stuck or a value.

Any λ_{cbv} expression represents a combinational circuit computing outputs instantaneously. Operators are predefined logic gates. Constants are bit vectors (*e.g.*, booleans are 1-bit vectors "0" and "1"). Each pair $(\mathring{v}_1, \mathring{v}_2)$ is a bit-vector concatenating the two projections $\mathring{v}_1$ and $\mathring{v}_2$. The *let* construct sequentially connects two sub-circuits. Each function abstraction **fun** $p \rightarrow e$ is a circuit with one input p and one output that is the return value of the function. The input (*resp.* the output) is composed of several wires: one for each bit of the argument value (*resp.* the result). Function application is circuit instantiation, *i.e.*, inlining, which eliminates parametric polymorphism and higher-orderness.

Fig. 2 gives an example of circuit description in λ_{cbv}. This is a classical 1-bit full adder sequentially connecting two instances of a half adder.

```
1  fun ((a,b),ci) ->
2    let half_add = fun (a,b) ->
3      (a xor b, a & b) in
4    let (s1,c1) = half_add(a,b) in
5    let (s,c2) = half_add(s1,ci) in
6    let co = c1 or c2 in
7    (s,co)
```

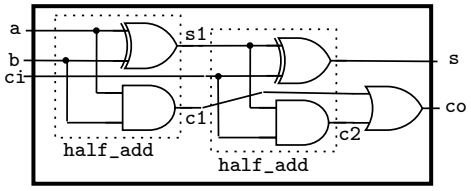


Fig. 2 1-bit full-adder description in λ_{cbv} and the corresponding circuit

2.2 A synchronous general purpose language

The λ_{cbv} language has limited expressiveness. It cannot describe, in particular, *sequential* circuits, *i.e.*, circuits having an internal state. Such circuits are commonly described by hardware designers using Moore or Mealy *Finite State Machines* encoded using low-level patterns at the register transfer level. Following a different path, Eclat extends λ_{cbv} in a way that allows compute-oriented sequential circuits to be directly expressed as parallel algorithms, while keeping control over the low-level timing of these circuits.

The upper part of Fig. 3 defines the syntax of Eclat as an extension of that of λ_{cbv}. It features two general purpose programming constructs, namely tail-recursion[5] (*fix*) and parallel pair $(e_1 \| e_2)$, both for which evaluation may span

[5] A recursive function **fix** f (**fun** $p \rightarrow e$) is a function f in which f is bound to its own definition **fun** $p \rightarrow e$ (*i.e.*, f can occur in e). In source programs, we use the classical derivated construct **let rec** $f\ p = e$ **in** e' defined as **let** $f =$ **fix** f (**fun** $p \rightarrow e$) **in** e' (see Fig. 4).

several cycles of a synchronizing clock. This notion of *clock* is incorporated in the language through the concept of *paused* expressions, which occur during evaluation (and never appear in source programs).

The lower part of Fig. 3 defines the operational semantics of ECLAT by an evaluation relation $e/\mu \Downarrow_n v/\mu'$ meaning: "*the configuration* e/μ *evaluates to* v/μ' *in exactly* n *clock cycles (with* $n \geq 0$*)*". This big-step relation $\Downarrow_n$ is defined by two rules, **EVAL-VAL** and **EVAL-PAUSE**, which *instantaneously* apply a sequence of zero or more reductions $\longrightarrow^\star$: if the redex is a value, evaluation terminates; if it is a *pause* of the form **pause** e', then the execution time is increased by 1 and the reduction of e' continues at the next cycle (in this case, we say that the evaluation is *non-instantaneous*).

Applying a recursive function f binds the argument to the formal parameter of f, unfolds the fixpoint and places a pause behind the function body (rule FIX-APP). Reducing a parallel pair $(e_1 \| e_2)$ is reducing e_1 (by applying rule CONTEXT in the context $(\Box \| e)$ until reaching either a paused expression or a value), then reducing e_2 in the same clock cycle (by applying rule CONTEXT in contexts $(v \| \Box)$ or $(\textbf{pause}\ e' \| \Box)$). Pauses are moved up by applying rule PAUSE, except for two special cases aiming to synchronize parallel pairs (rules PAUSE-PAR1 and PAUSE-PAR2). Once both e_1 and e_2 have been reduced to values, a pair is instantaneously created (rule PAR).

expression	$e ::= \cdots \mid \textbf{pause}\ e \mid \textbf{fix}\ f\ (\textbf{fun}\ p \to e) \mid (e \| e)$
value	$v ::= \cdots \mid \textbf{fix}\ f\ (\textbf{fun}\ p \to e)$

$$\textbf{EVAL-VAL}\ \frac{e/\mu \longrightarrow^\star v/\mu'}{e/\mu \Downarrow_0 v/\mu'} \qquad \textbf{EVAL-PAUSE}\ \frac{e/\mu \longrightarrow^\star \textbf{pause}\ e'/\mu' \quad e'/\mu' \Downarrow_n v/\mu''}{e/\mu \Downarrow_{(n+1)} v/\mu''}$$

evaluation context $F ::= \cdots \mid (\Box \| e) \mid (v \| \Box) \mid (\textbf{pause}\ e \| \Box)$

FIX-APP $\quad (\underbrace{\textbf{fix}\ f\ (\textbf{fun}\ p \to e)}_{\phi})\ v/\mu \longrightarrow \textbf{pause}\ ((e[f \mapsto \phi])[p \mapsto v])/\mu$

PAUSE $\quad F[\textbf{pause}\ e]/\mu \longrightarrow \textbf{pause}\ F[e]/\mu \quad \text{si} \begin{cases} F \not\equiv (\Box \| e') \\ F \not\equiv (\textbf{pause}\ e' \| \Box) \end{cases}$

PAUSE-PAR1 $\quad (\textbf{pause}\ e \parallel v)/\mu \longrightarrow \textbf{pause}\ (e \| v)/\mu$

PAUSE-PAR2 $\quad (\textbf{pause}\ e \parallel \textbf{pause}\ e')/\mu \longrightarrow \textbf{pause}\ (e \| e')/\mu$

PAR $\quad (v \| v')/\mu \longrightarrow (v, v')/\mu$

Fig. 3 Syntax and reduction semantics of ECLAT as an extension of λ_{cbv} (Fig. 1)

Let $\phi_{\mathtt{add}}$ be: **fix** `add` **(fun** `(a,b)` `->` **if** `a = 0` **then** `b` **else** `add(a-1,b+1))`. Evaluating $\phi_{\mathtt{add}}\,\mathtt{(2,2)}/\mu$ for a given memory μ behaves as follows:

at cycle 0: $\phi_{\mathtt{add}}\,\mathtt{(2,2)}/\mu \longrightarrow^\star \textbf{pause}\ e_0/\mu$ where $e_0 =$ **(if** `2 = 0` **then** `2` **else** $\phi_{\mathtt{add}}\,\mathtt{(2-1,2+1))}$, and then the pause is removed (by **EVAL-PAUSE**);

at cycle 1: $e_0/\mu \longrightarrow^\star \textbf{pause}\ e_1/\mu$ where $e_1 =$ **(if** `1 = 0` **then** `3` **else** $\phi_{\mathtt{add}}\,\mathtt{(1-1,3+1))}$, and then the pause is removed (by **EVAL-PAUSE**);

at cycle 2: $e_1/\mu \longrightarrow^\star \mathtt{4}/\mu$; therefore $\phi_{\mathtt{add}}\,\mathtt{(2,2)}$ evaluates to `4` in 2 cycles.

Fig. 4 illustrates how behaves $(e_1 \| e_2)$. At lines 1-5 is defined a function `collatz` computing the stopping time of the *Collatz* sequence (also referred to as Syracuse) starting from an integer `n`. It declares a local tail-recursive function `f` performing one call per cycle. At line 6, function `collatz` is called twice in parallel with arguments `4` and `8`. It evaluates as follows:

at cycle 0: `collatz(4)` reduces to `f(4,0)` and then `collatz(8)` to `f(8,0)`;
at cycle 1: `f(4,0)` reduces to `f(2,1)` and then `f(8,0)` reduces to `f(4,1)`;
at cycle 2: `f(2,1)` reduces to `f(1,2)` and then `f(4,1)` reduces to `f(2,2)`;
at cycle 3: `f(1,2)` reduces to `2` and then `f(2,2)` reduces to `f(1,3)`;
at cycle 4: `f(1,3)` reduces to `3` and then the program returns `2 + 3`.

```
1 let collatz n =
2   let rec f (n,t) =
3     if n = 1 then t else
4     if n mod 2 = 0 then f(n/2,t+1) else f(3*n+1,t+1)
5   in f(n,0) in
6 let (x,y) = (collatz(4) || collatz(8)) in x + y
```

Fig. 4 A parallel computation expressed in Eclat

Note that in $(e_1 \| e_2)$, e_1 and e_2 are reduced *sequentially* (from left to right) within the same clock cycle. This provides a deterministic *parallel evaluation* of e_1 to a value v_1 and e_2 to v_2, building the pair (v_1, v_2). If expressions e_1 and e_2 are instantaneous, then expressions (e_1, e_2) and $(e_1 \| e_2)$ are equivalent.

2.3 Compilation

Due to space limitation, the compilation process turning Eclat programs into circuits cannot be presented in detail in this paper. But having a simplified view of the compiler ease to estimate the size of the resulting hardware. The basic principles of the compilation scheme can be summarized as follows:

1. lexical environments are made explicit by additional function parameters and then functions are globalized (by λ-*lifting*);
2. higher-order functions (*resp.* polymorphic functions) are specialized (*resp.* monomorphized) and therefore duplicated.
3. non-recursive functions are systematically inlined,
4. tail-recursive function calls are shared[6] (*i.e.,* not duplicated) except for parallel calls (*e.g.*, Fig. 4, line 6), which are duplicated.
5. compiling construct $(e_1 \| e_2)$ leads to two automata for e_1 and e_2, which are local to the current state. The whole program becomes the (combinational) transition function of a Moore machine[7] driven by the global clock.

[6] Each tail-recursive function definition becomes a state in an underlying hierarchical automaton. Calling a tail-recursive function f becomes a transition to the corresponding state (this implements *pauses* from the semantics).

[7] Eclat provides circuit parallelism, with no extra cost for synchronization (no lost cycle). The reduction order $(v \| \square)$ for parallel branches in the semantics is convenient for reasoning on the behavior of programs and it does not impact performance.

Consider a sequence **let** $x = e_1$ **in** e_2 where e_1 and e_2 are instantaneous. If there is a data dependency between e_1 and e_2 (*i.e.,* if x does occur in e_2), a wire x connects e_1 to e_2 and therefore, the critical path (*i.e.,* the Worst Case Execution Time) could increase, but the design remains synthesizable up to a limit imposed by the frequency of the physical clock. If there is no data dependency between e_1 and e_2, both can be implemented as parallel hardware.

General recursion is not supported because it would limit scalability and exploitable parallelism; but it can be encoded by the programmer using explicit stacks implemented as vectors or arrays, the later being presented in sec. 3.

3 Mixing synchrony and shared memory

Hardware description languages such as VHDL support array types. Synthesis tools implement such arrays either as collections of logic elements or using on-chip RAM blocks. The former is only applicable to arrays of small size because it consumes a large number of logic elements both for storage and for the addressing circuitry. The latter requires that the source code is written using a specific pattern to be correctly recognized and handled by the synthesizer. This pattern induces extra clock cycles to access data stored in the RAM blocks.

This section presents how mutable arrays can be formalized in Eclat to be implemented using RAM blocks. Eclat takes advantage of the underlying synchronous model to safely support concurrent memory accesses, in a predictable and fair way, while exploiting physical parallelism at the circuit level.

3.1 Mutable arrays in Eclat

The array operations provided in Eclat are listed in the upper part Fig. 5. These operations come with a few Eclat features that have not been presented in the last section (Fig. 1 and 3). This includes a type constructor $\mathring{\tau}$ `array<`s`>`, which represents arrays of length s with elements of basic type $\mathring{\tau}$.

Operator `create<`s`>` (of type scheme $\forall\mathring{\tau}\cdot$ `unit` $\to \mathring{\tau}$ `array<`s`>`) creates an array of length n with unspecified values of a given type $\mathring{\tau}$. Operator `length` instantaneously returns the length of its array argument. Operators `get` and `set` allow for reading and modifying arrays. Each array creation returns a location ℓ, which is a value, associated with a data structure located in memory. Each location is protected by a lock, which is explicitly manipulated by the `%acquire` and `%release` primitives. No location nor *acquire*/*release* operations appear in source programs; these are hidden in the definitions of `get` and `set`. Arrays are not immediate values (*i.e.*, constants and pairs of immediate values) unlike vectors (of basic type $\mathring{\tau}$ `vect<`s`>`) presented in sec. 2.1.

In concrete syntax, we simply write `create` e by letting the compiler statically evaluate a simple arithmetic expression e to an integer for inferring the length of the array to be created. The sequence e_1 `;` e_2 stands for **let** `_` $= e_1$ **in** e_2 with `_` a wildcard pattern. The macro expression **parfor** $x = e_{\mathsf{a}}$ **to** e_{b} **do** e **done** builds the expression **let** `_` $= (e_i \| \cdots \| e_j)$ **in** `()` with e_{a} (*resp.* e_{b}) an arithmetic expression statically evaluated to i (*resp.* j) and $e_k \stackrel{\text{def}}{=} (\textbf{let}\ x = k\ \textbf{in}\ e)_{k\in\{i,\cdots\ j\}}$.

operator	op	::=	$\cdots$ | `create<`s`>` | `length` | `get` | `set` | `%acquire` | `%release`
expression	e	::=	$\cdots$ | ℓ
value	v	::=	$\cdots$ | ℓ
type	τ	::=	$\cdots$ | $\mathring{\tau}$ `array<`s`>`

ϑ-Create	$\vartheta($`create<`n`>`$,(),\mu) = \ell/\mu[\ell \mapsto$ `(false,(`n`,(`$v_0,\cdots v_{n-1}$`)))`$]$ where ℓ is a fresh location and $v_0,\cdots v_{n-1}$ are unspecified values
ϑ-Length	$\vartheta($`length`$,\ell,\mu) = n/\mu$ if $\mu(\ell) =$ `(`b`,(`n`,`v`))`
ϑ-Get-wait	$\vartheta($`get`$,(\ell$`,`$n),\mu) =$ (**pause** `();` `get(`ℓ`,`n`))`$/\mu$ if $\mu(\ell) =$ `(true,`v`)`
ϑ-Set-wait	$\vartheta($`set`$,((\ell$`,`$n)`,`$v'),\mu) =$ (**pause** `();` `set((`ℓ`,`n`),`v'`))`$/\mu$ if $\mu(\ell) =$ `(true,`v`)`
ϑ-Get	$\vartheta($`get`$,(\ell$`,`$i)`,`$\mu) =$ (`%acquire (`ℓ`);` **pause** `();` **pause** `();` `%release(`ℓ`);` v_i`)`$/\mu$ if $\mu(\ell) =$ `(false,(`n`,(`$v_0,\cdots v_i,\cdots v_{n-1}$`)))`
ϑ-Set	$\vartheta($`set`$,((\ell$`,`$i)`,`$v_i'),\mu) =$ (`%acquire(`ℓ`);` **pause** `();` **pause** `();` `%release(`ℓ`))`$/\mu'$ where $\mu' = \mu[\ell \mapsto$ `(false,(`n`,(`$v_0,\cdots v_{i-1}$`,`v_i'`,`$v_{i+1},\cdots v_{n-1}$`)))`$]$ if $\mu(\ell) =$ `(false,(`n`,(`$v_0,\cdots v_{i-1},v_i,v_{i+1},\cdots v_{n-1}$`)))`
ϑ-Acquire	$\vartheta($`%acquire`$,\ell,\mu) =$ `()`$/\mu[\ell \mapsto$ `(true,`v`)`$]$ if $\mu(\ell) =$ `(false,`v`)`
ϑ-Release	$\vartheta($`%release`$,\ell,\mu) =$ `()`$/\mu[\ell \mapsto$ `(false,`v`)`$]$ if $\mu(\ell) =$ `(true,`v`)`

Fig. 5 Syntax and semantics of array operations (as an extension of Fig. 1 and 3)

The behavior of array operations in Eclat is formalized on the lower part of Fig. 5. It is based on the operational semantics of Eclat (previously defined in Fig. 1 and 3) and mainly consists in refining the semantic function ϑ used to specify, by cases, the behavior of each Eclat operator.

A memory μ (kept abstract in the last section) is a partial function from memory locations to values. Memory locations are denoted with meta-variable ℓ. We note $\mu(\ell)$ (*resp.* $\mu[\ell \mapsto v]$) the value stored at location ℓ in μ (*resp.* the new memory obtained by assigning a value v to location ℓ in μ).

Array creation returns a location ℓ mapped to a value representing an array in the memory (rule ϑ-Create). In the semantics, this value is encoded as two nested pairs `(`*lock*`,(`N`,`*content*`))` where:

- *lock* is a boolean indicating if the lock associated with the array is held,
- N is the length of the array,
- *content* represents the N elements of the array as a tuple. For example, an array of integers will be encoded as `(false,(4,(0,0,0,0)))`.

The role of the two pauses performed by `get` and `set` (cases ϑ-Get and ϑ-Set) is to accurately reflect the behavior of the hardware implementation (a synchronous Moore machine) that is generated by the Eclat compiler[8].

Concurrent memory accesses are safe because of the lock mechanism, which is specified by cases Acquire and Release. When the lock associated with location ℓ is already held, any other access to ℓ is delayed until the lock is released (cases ϑ-Get-Wait and ϑ-Set-Wait). As there is one lock per array, parallel accesses to different arrays are executed "simultaneously", *i.e.*, in the same sequence of clock cycles. For instance, the expression defined on the left side of Fig. 6 behaves as depicted in the chronogram on the right side:

[8] The address of the element to be read in the RAM block is set *at next clock tick*. Then, the RAM block (which also behaves as a Moore machine) makes the read value available one clock cycle later.

at cycle 0: after sequentially creating the two arrays, the left branch of the parallel (`x`) acquires the lock associated with array `a` (which is noted $\mathcal{A}_a$ in the chronogram) and starts a sequence of two pauses (which are noted "-") as specified by ArraySet in Fig. 5; then, the right branch (`y`) does the same with array `b` ($\mathcal{A}_b$);
at cycle 1: `x` performs its second pause, then `y` performs its second pause;
at cycle 2: `x` releases the lock of `a` ($\mathcal{R}_a$); then `y` releases the lock of `b` ($\mathcal{R}_b$).

```
1  let a = create 100 in
2  let b = create 100 in
3  let (x,y) = (set((a,0),42) || set((b,0),43)) in ()
```

cycles:	0	1	2
x:	$\mathcal{A}_a$; -	-	$\mathcal{R}_a$
y:	$\mathcal{A}_b$; -	-	$\mathcal{R}_b$

Fig. 6 Eclat expression simultaneously executing two independent array accesses

3.2 Concurrent programming

The semantics of Eclat provides the programmer with a predictable mechanism for dealing with concurrency:

- *left-to-right reduction order*: if two expressions e_{left} and e_{right} start at the same cycle and are put in parallel (*e.g.*, $(e_{\mathsf{left}} \| e_{\mathsf{right}})$), then e_{left} always reduces before e_{right} (though the implementation, which statically encodes this reduction order in synchronous dataflow style, is intrinsically parallel);
- *atomicity of sequential composition*: if an access to ℓ is followed by expression e, then e always reduces before any concurrent access waiting for ℓ.

For example, there is no data race in the program defined in Fig. 7. We have the guarantee that it always evaluates to `43`, as illustrated in the execution trace:

at cycle 0: the array is created; then the first binding (`x`) acquires the lock and pauses (`-`); then the second binding (`y`) tries to acquire the lock but cannot (✗) because the lock is already held by `x`.
at cycle 1: `x` is paused (the array write progresses); then `y` is reduced but it has to wait because the lock is already held by `x`.
at cycle 2: `x` releases the lock and *atomically continues as far as possible*: it acquires the lock again and pauses; then `y` has to wait because the lock is already held by `x`.
at cycle 3: `x` is paused (the array read progress); then `y` has to wait because the lock is already held by `x`.
at cycle 4: `x` releases the lock and returns value `44`; then `y` acquires the lock and pauses.

Note that this strict ordering of operations does not prevent the programmer from writing programs whose behavior cannot be predicted statically. For example, in the following expression, the writing order (and thus the result) depends on the duration of the two calls to `collatz` (defined in Fig. 4):

```
let _ = ( (collatz(i); set((a,0),1))
        || (collatz(j); set((a,0),42)) ) in get(a,0)
```

```
1  let a = create 1 in
2  let (x,y) = ((set((a,0),42); set((a,0),43); 44) || get(a,0)) in y
```

cycles:	0	1	2	3	4	5	6
x:	$\mathcal{A}_a$; -	-	$\mathcal{R}_a$;$\mathcal{A}_a$ -	-	$\mathcal{R}_a$		
y:	✗	✗	✗	✗	$\mathcal{A}_a$; -	-	$\mathcal{R}_a$

Fig. 7 Eclat program and its execution trace with atomic sequential composition

The trade-off between expressiveness and predictability therefore remains in the hand of the programmer.

When programming in Eclat, one might question the fairness of the mechanism for concurrent memory accesses. The lock mechanism of Eclat is intrinsically fair with regard to the following criterion: let n parallel expressions $(e_1 \parallel e_2 \parallel \cdots e_n)$ performing concurrent memory accesses and $i < j < k < n$. If e_j holds a lock and both e_i and e_k wait for this lock, then e_k will always take the lock before e_i. This comes from the *left-to-right reduction order* property (sec. 3.2) because at each cycle, e_i is reduced before e_j, and thus before e_j releases the lock, which is then available when reducing e_k.

This is exemplified with the program defined in Fig. 8. Each time an array access starts, it acquires the lock (A_a) for two cycles until the lock is released (R_a). Any other access to `a` while the lock is hold is delayed. This directly come from the behavior of the Eclat parallel construct $(e_1 \| e_2)$: at each clock cycle, e_1 is instantaneously reduced until it becomes a pause or a value, then e_2 is instantaneously reduced until it becomes a pause or a value, and there is a synchronization at the end of the cycle, resulting in a deterministic parallelism. The chronogram in Fig. 8 gives the corresponding execution trace. An interesting point occurs at cycles 7-8:

at cycle 7: *no array access is executed*; indeed, `x` *cannot* acquire the lock because `z` already holds it since cycle 5; then `y` *cannot* acquire the lock; then `z` releases the lock and continues with `f(2)`;

at cycle 8: `x` acquires the lock (which has been released by `z` at cycle 7) and pauses; then `y` *cannot* acquire the lock: this is exactly the same configuration as at cycle 1.

```
1  let a = create 3 in
2  let rec f (i) = set((a,i),i); f(i) in (* loops forever *)
3  let ((x,y),z) = ( (f(0) || f(1)) || f(2) ) in ()
```

cycles:	0	1	2	3	4	5	6	7	8	9	
x:	f(0)	$\mathcal{A}_a$; -	-	$\mathcal{R}_a$;f(0)	✗	✗	✗	✗	$\mathcal{A}_a$; -	-	···
y:	f(1)	✗	✗	$\mathcal{A}_a$; -	-	$\mathcal{R}_a$;f(1)	✗	✗	✗	✗	···
z:	f(2)	✗	✗	✗	✗	$\mathcal{A}_a$; -	-	$\mathcal{R}_a$;f(2)	✗	✗	···

Fig. 8 Fair interlacing of concurrent memory accesses in an Eclat program

Note that construct `(`$e \| e'$`)` is not commutative (this is a specificity of ECLAT), but is associative. Reducing $((e_1 \| e_2) \| e_3)/\mu$ is reducing e_1/μ to e_1'/μ_1 and e_2/μ_1 to e_2'/μ_2, and then e_3/μ_2 to e_3'/μ_3. Reducing $(e_1 \| (e_2 \| e_3))/\mu$ is reducing e_1/μ to e_1'/μ_1, and then e_2/μ_1 to e_2'/μ_2 and e_3/μ_2 to e_3'/μ_3.

4 Evaluation

This section illustrates the ability to exploit in ECLAT both shared memory and fine-grained parallelism with significant performance gains compared to sequential code running on a CPU. It also shows how the programmer can easily estimate timing performance from ECLAT source code.

Experimental setup We target an Intel Max10 FPGA embedded on a Terasic DE10-Lite board. This circuit has a frequency of 50 MHz; it comprises 50K logic elements (LEs) and 1,638 Kbits of on-chip RAM blocks. Each ECLAT program is compiled to VHDL using the ECLAT compiler. All of the ECLAT programs presented here meet the timing requirements of the target (*i.e.*, they can safely run using the global 50 MHz clock). The generated VHDL code is synthesized using the Quartus 22.1 Lite toolchain, which indicates resource usage (*i.e.*, both the number of LEs and the number of RAM blocks used by the design). Execution time estimation is measured both by software-RTL simulation using GHDL[9] and synthesis. Speedups are given with regard to sequential OCAML code running on an Intel core i7 CPU with 16 GB of RAM and a frequency of 2.2 GHz (*i.e.*, 44 times faster than the global clock of the Max10 FPGA). OCAML code is compiled to native code using the OCAML compiler `ocamlopt` with optimizations enabled (`-O3`).

4.1 *Game of life* - first version

Fig. 9 is an implementation of *Conway's Game of Life (GoL)* using ECLAT. Note that this ECLAT program is also a valid OCAML program. Function `array_life` (lines 22-28) computes the next generation of a world of cells (being either *alive* or *dead* depending on the number of alive neighboring cells). The world is represented as an array of boolean cells. The higher-order function `array_mapi` (lines 15-20) has type $\forall\eta \cdot \forall\mathring{\alpha} \cdot \forall\mathring{\beta} \cdot$ `(((int` $\times\, \mathring{\alpha}$`)` $\to \mathring{\beta}$`)` $\times\mathring{\alpha}$ `array<`η`>` $\times\, \mathring{\beta}$ `array<`η`>)` $\to$ `unit`. It applies a function `f` to each element of a source array `src` and fills a destination array `dst`.

Function `array_mapi` performs a direct call to the local recursive function `aux` (taking one cycle), and then one read (2 cycles) plus one write (2 cycles) plus a recursive call (1 cycle) for each of the N iterations. The resulting execution time is $1 + \sum_{i=0}^{N-1}(2 + T(\mathtt{f}_i) + 2 + 1)$ where $T(\mathtt{f}_i)$ is the execution time of `f` applied to the i-th array element.

For processing one cell (*i.e.*, reading it and modifying it in memory), the eight neighbors have also to be read, resulting in a throughput of $5 + 8 \times 2 = 21$ cycles per cell. When synthesized on the Max10 FPGA (clocked at 50 MHz),

[9] http://ghdl.free.fr

```
1  let neighborhood(f,i,n) =
2    f(i-n-1) + f(i-n) + f(i-n+1) +
3    f(i-1)            + f(i+1) +
4    f(i+n-1) + f(i+n) + f(i+n+1) ;;
5
6  let next_cell(get_cell,w,i,cell,n) =
7    let alive_int(i) =
8      if get_cell(w,i) then 1 else 0 in
9    let s = neighborhood(alive_int,i,n)
10   in (cell & s = 2) or (s = 3) ;;
11
12 let pos_modulo(i,n,size) =
13   if i < 0 then i+size else
14   if i >= size then i-size else i ;;
```

```
15 let array_mapi (f,src,dst) =
16  let rec aux(i) =
17    if i < length(src) then
18    (set((dst,i),f(i,get(src,i)));
19     aux(i+1)) else ()
20  in aux(0) ;;
21
22 let array_life (src,dst,n) =
23  let access(w,i) =
24   get(w,pos_modulo(i,n,length w))
25  in
26  let f (i,cell) =
27   next_cell(access,src,i,cell,n)
28  in array_mapi(f,src,dst) ;;
```

Fig. 9 Game of Life, version 1 – the world is an array of booleans (shared memory)

`array_life` is 1.3 times faster than the same implementation (seen as an OCaml function) compiled to native code and running[10] on a CPU (clocked at 2.2 GHz). This measure must take into account the difference in both frequency and memory capacity of the FPGA (limited to RAM blocks) with regard to the CPU. However, we could expect a better speedup because this first implementation does not exploit the parallelism provided by the target FPGA.

4.2 Game of Life - version 2

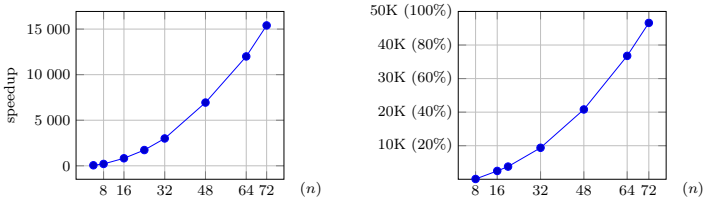
Bakhteri *et al.* [3] have presented a Verilog implementation of GoL, which is able to process the entire world within one clock cycle. Eclat is expressive enough to describe such an implementation. For this, the function `array_life` of Fig. 9 must be replaced by the function `vect_life` given in Fig. 10, which now operates on a world represented as a single vector of boolean values (*i.e.*, an immediate value, similar to a very large integer). Function `next_cell` is defined in Fig. 9 (lines 6-10) and called point-by-point on the vector in Fig. 10 (lines 30-31). These calls duplicate the body of `next_cell` for each cell in the world, so that they can be computed in parallel within one cycle. The primitive `vect_mapi` (presented in Fig. 2) is similar to the function `array_mapi` defined Fig. 9, but now operates instantaneously on vectors.

```
29 let vect_life (world,n) =
30   let access(w,i) = vect_nth(w,pos_modulo(i,n,vect_size w)) in
31   let f (i,cell) = next_cell(access,world,i,cell,n) in
32   vect_mapi(f,src) ;;
```

Fig. 10 Game of Life, version 2 – the world is a boolean vector (*i.e.*, an immediate value)

[10] The entry point of the Eclat (*resp.* OCaml) program creates the `src` and `dst` arrays, then applies `array_life` hundreds of times (using a loop). This makes array allocation and initialization negligible in both Eclat and OCaml. Furthermore RTL simulation confirms the accurate execution time prediction for `array_life` in Eclat (*i.e.*, 21 cycles per cell).

Fig 11 compares the performance of `vect_life` on FPGA for a world of size n^2 to that obtained with the previously mentioned `array_life` running as an OCaml program on a CPU. Fig. 11a gives the speedup and Fig. 11b the number of LEs and the corresponding percentage of LE usage (up to 50K LEs on the Max10 FPGA). The curves perfectly illustrate the kind of time-space trade-off that can be obtained by using Eclat. The maximum observed speedup is significant: $\times 15,000$ for a world of $5,000$ cells. Beyond this limit, the design does not fit in the Max10 FPGA board we use[11].



(a) speedup wrt CPU execution (b) number of LEs

Fig. 11 Time-space trade-off in the `vect_life` implementation of GoL

4.3 Game of life - version 3

To overcome the size limitation of the fine-grain parallel version described in sec. 4.2, a possible solution is to represent the world not as a single vector but as an *array* of vectors, each vector representing a line. With this formulation, lines will be stored in shared memory, and fine-grained parallelism can be exploited to process individual lines.

The corresponding implementation, named `vect_array_life`, is given in Fig. 12. Note that function `next_cell_with_lines(j,cell,line0,line1, line2)` (which is called in Fig. 12, line 5, but not given by lake of space) is very similar to function `next_cell` (line 6-10, Fig. 9): it determines if the current `cell` located at index j in line `line1` becomes dead or alive in the next generation, knowing the neighboring lines (`line0` and `line2`).

For a world of N lines, `vect_array_life` reads $N+1$ lines, independently of the number W of cells per lines. The transformation is performed in-place, there are N array writes. The overall execution time is exactly $(2+2+1)N+1 = 5N+1$ cycles to process an entire world of size $N \times W$. The execution time and the number of RAM blocks grows linearly with N, while the number of LEs grows with W exactly as in Fig. 11b. Note that `vect_array_life` is 10 times slower than `vect_life` for a same number of cells per line, but the world it processes can be 200 times larger.

Fig. 13 summarizes the performances (for both execution time and LEs/RAM usage) of all versions of GoL described in this section.

[11] High-end FPGAs have much more LEs, *e.g.*, 2.8 million LEs in the Intel Stratix 10 (*i.e.*, 56 times more than the Max10).

```
1  let vect_array_life (world : bool vector<'a> array<'b>) : unit =
2    let first_line = get(world,0) in
3    let rec aux (line0,line1,i) : unit =
4       if i < length(world) then
5        (let line2 = if i = length(world)-1 then first_line else get(world,i+1) in
6         let next (j,cell) = next_cell_with_lines(j,cell,line0,line1,line2) in
7         set((world,i),(vect_mapi(next,line1)));
8         aux(line1,line2,i+1)) else ()
9    in aux(get(world,length(world)-1),first_line,0) ;;
```

Fig. 12 GoL, version 3 – the world is an array of vectors (*i.e.*, a vector per line)

implementation	largest size	LEs	RAM	throughput
Bakhteri *et al.* [3] (Verilog)	64×64	68K	none	4,096 cells each cycle
GoL, v2 (`vect_life`)	72×72	47K	none	5,184 cells each cycle
GoL, v1 (`array_life`)	$1,024 \times 512$	2K	1Mbit	1 cell each 21 cycles
GoL, v3 (`vect_array_life`)	$2,048 \times 512$	49K	1Mbit	2,048 cells each 5 cycles

Fig. 13 Space-time trade-off for different GoL implementations

5 Developing parallelism skeletons with Eclat

This section presents parallel implementations of *map* and *pipe* algorithmic skeletons using Eclat. It focusses on composability and gives an example of performance prediction, experimentally corroborated.

The `map` skeleton is classically used to apply a given function `f` to each element of an array. A parallel implementation of `map`, called `par_map`, is given in Fig 14 (lines 12-18). It takes a slice length `p`, a function `f`, a source array `src`, and a destination array `dst`. It applies `f` to the elements of `src`, traversing `p` slices of the source array in parallel (a call to `map_slice` per slice) using the macro-construct **parfor** presented in sec. 3. If the length of `src` is not a multiple of `p`, the remaining elements are processed sequentially (line 18). The result array `dst` must be passed as an argument since, as already indicated, Eclat functions cannot return arrays (which are not immediate values).

```
1  let map_slice(a,b,f,src,dst) =
2    let rec aux(i) =
3      if i < b then
4        (let x = get(src,i) in
5         let y = f(x) in
6         set((dst,i),y);
7         aux(i+1))
8      else ()
9    in
10   if a >= b then ()
11   else aux(a) ;;
```

```
12 let par_map(p,f,src,dst) =
13   let n = length src in
14   let d = n/p in
15   parfor i = 0 to p-1 do
16     map_slice(d*i,d*(i+1),f,src,dst)
17   done;
18   map_slice(d*p,n,f,src,dst) ;;
19
20 let map((p,f,src),k) =
21   let dst = create (length src) in
22   par_map(p,f,src,dst); k(dst) ;;
```

Fig. 14 A parallel implementation of `map`

Let us add a new notation $f\ e_1$ `@@` e_2 that stands for f`(`e_1`,`e_2`)`. Function `map` (lines 20-22) is a composable extension of `par_map`, avoiding passing the result array explicitly. For this, `map` takes (in addition to `p`, `f` and `src`) a *continuation function* `k`. It locally allocates a destination array `dst` (line 21), calls `par_map(p,f,src,dst)` and then applies `k` to `dst`. For example, applying two maps in sequence with function `f` and then `g` (for a given degree of parallelism `p`) is expressible as: `map(p,f,src) @@ fun a -> map((p,g,a),k)`.

In Fig. 15 is defined another parallel implementation `map_farm`, using a so-called *worker farm* for applying *on request* the `f` function to the elements of the `src` array. This is well-suited when `f` has an irregular execution time depending on the elements of `src`. Workers communicate via a shared variable `r`, which is an array of length 1 containing the index of the next array element to be processed. Each time a worker is available, it: **(1)** reads the index given by `r`; **(2)** increments `r`; **(3)** processes one element, this until the integer `i` referenced by `r` reaches the end of the source array. The important point is that steps **(1)** and **(2)**, *i.e.,* reading and writing `r` (lines 3-5), are always performed atomically (as explained in sec. 3.2).

```
1  let worker (f,r,src,dst) =
2    let rec loop() =
3      let i = get(r,0) in
4      if i >= length src then () else
5        let ((),x) = (set((r,0),i + 1)
6                      || get(src,i)) in
7        (set(dst,i,f(x)); loop())
8    in loop() ;;
```

```
9  let map_farm((p,f,src),k) =
10   let r = create 1 in
11   set((r,0),0);
12   let dst = create (length src) in
13   parfor _ = 0 to p-1 do
14     worker(f,r,src,dst)
15   done;
16   k(dst) ;;
```

Fig. 15 Parallel implementation of `map` using a worker farm

In Fig. 16 is defined a `pipe3` skeleton. Functionally, `pipe3((f1,f2,f3,src),k)` is equivalent to `map(1,f1,src) @@ fun a1 -> map(1,f2,a2) @@ fun a2 -> map((1,f3,a2),k)`, but these successive `map`s (or *stages*) are parallelized. The implementation described here could easily be generalized by hand to achieve more parallelism (*i.e.*, more stages). On the left side of Fig. 16 is defined a function `semi_pipe`, which consumes the elements x_i of `src` if their position i is less than the index referenced by `rI` and otherwise actively waits. Function `f` is applied to each x_i consumed. Then, `semi_pipe` assigns the resulting value in `dst` (line 9) and increments the shared variable `r0` (line 10) to notify that `dst` is ready to be processed up to index i. On the right side of Fig. 16, `pipe3` creates intermediate arrays and shared variables (the latter being initialized to 0, except for `r0`, which takes value `n = length src` to notify that `src` is ready to be consumed), calls the `semi_pipe`s in parallel, and then `k`.

Performance prediction A key aspect of the synchronous semantics of ECLAT is the ability to predict the execution time of parallel programs. Let's illustrate this with an example. The `par_map` skeleton (defined in Fig. 14) calls `map_slice(a,b,f,src,dst)`, `p` times in parallel. Each of these calls comprises

```
let semi_pipe(f,(rI,src),
              (r0,dst)) =
  let rec loop(i) =
    if i < length src then (
      let (j,x) = (get(rI,0)
                || get(src,i))
      in if i < j then
           let y = f x in
           set((dst,i),y);
           set((r0,0),i+1);
           loop(i+1)
         else (loop(i))) else ()
  in loop(0)  ;;
```

```
let pipe3 ((f1,f2,f3,src),k) =
  let n = length src in
  let r0 = create 1 in
  let (a1,r1) = (create n, create 1) in
  let (a2,r2) = (create n, create 1) in
  let (a3,r3) = (create n, create 1) in
  let _ = (set(r0,0,n) || set((r1,0),0)
    || set((r2,0),0) || set((r3,0),0)) in
  let _ =
    ((semi_pipe(f1,(r0,src),(r1,a1))
    || semi_pipe(f2,(r1,a1),(r2,a2)))
    || semi_pipe(f3,(r2,a2),(r3,a3)))
  in k(a3) ;;
```

Fig. 16 Eclat implementation of a three-stage pipeline

one direct call to function `aux` (taking one clock cycle) followed by one read (2 cycles), one call to `f`, one write (2 cycles) and one tail-recursive call (1 cycle) per iteration. Let $T(\mathtt{f}_i)$ be the execution time of `f` when applied to the i-th element of array `src` and let N b the length of `src`. The execution time of `map_slice` (in cycles) is:

$$T_{\mathtt{map\_slice}}(\mathtt{a},\mathtt{b}) = 1 + \sum_{i=\mathtt{a}}^{\mathtt{b}-1} \Big(2 + T(\mathtt{f}_i) + 2 + 1\Big)$$

For the sake of simplicity, assume `f` is stateless[12] and N is a multiple of `p`. Due to the fairness of execution (explained in sec. 3.2), the execution time of `map_par` is the maximum execution time among each $T_{\mathtt{map\_slice}}$ plus an amount of time $T_{\mathtt{stalls}}$ (or *lost cycles*) spent in sequentializing concurrent accesses[13]:

$$T_{\mathtt{map\_par}} = \max_{i=0}^{\mathtt{p}-1} \Big(T_{\mathtt{map\_slice}}(i \times N/\mathtt{p}, (i+1) \times N/\mathtt{p})\Big) + T_{\mathtt{stalls}}$$

If the computation time dominates the access time, then $T_{\mathtt{stalls}}$ becomes negligible. Note that $T_{\mathtt{stalls}}$ is clearly less than $4N$, *i.e.*, the cost of sequentially traversing `src` (N reads) and updating `dst` (N writes). The chronogram in Fig. 17, gives the beginning of the execution of `map_par` for a regular computation such that $T(\mathtt{f}_i) = T_\mathtt{f}$ and $T_\mathtt{f} = 2\mathtt{p} - 4$. There are $2\mathtt{p} - 2$ stalls (✗) at initiation (since each call to `map_slice` simultaneously tries to read `src`, but only one acquires the lock ($\mathcal{A}_{\mathtt{src}}$) at a time. Note that in this specific case, after initiation, there are no stalls anymore. From this we can deduce that, for any stateless function `f` such that $T_\mathtt{f} \geq 2\mathtt{p}-4$, we have $T_{\mathtt{stalls}} = 2\mathtt{p}-2$ cycles. In other words, if $T_\mathtt{f} \geq 2\mathtt{p}-4$, then the computation of `f` dominates accesses.

This theoretical result is verified in practice. For example, for $N = 3,200$ and $\mathtt{p} = 16$ and `f` such that $T_\mathtt{f} = 2\mathtt{p} - 4 = 28$ cycles, the measured execution time of `par_map` is 6,631 cycles, *i.e.* $(1+\max_{i=0}^{\mathtt{p}-1}(2+28+2+1)N/\mathtt{p})+(2\times\mathtt{p}-2)$.

[12] The programmer is free to compose complex behaviors (*e.g.*, global array accesses from the function `f` to be parallelized): this remains deterministic, with a trade-off between expressiveness and predictability.

[13] Each `map_slice` reads `src` and write `dst` concurrently.

cycles:	0	1	2	3	4	5	···	2p-1	2p	2p+1	2p+2	
$\texttt{slice}_1$:	aux	$\mathcal{A}_{\texttt{src}}$; -	-	$\mathcal{R}_{\texttt{src}}$; f	-	-	···	$\mathcal{A}_{\texttt{dst}}$; -	-	$\mathcal{R}_{\texttt{dst}}$;aux	$\mathcal{A}_{\texttt{src}}$; -	···
$\texttt{slice}_2$:	aux	✗	✗	$\mathcal{A}_{\texttt{src}}$; -	-	$\mathcal{R}_{\texttt{src}}$; f	···	-	-	$\mathcal{A}_{\texttt{dst}}$; -	-	···
$\texttt{slice}_\texttt{p}$:	aux	✗	✗	✗	✗	✗	···	$\mathcal{A}_{\texttt{src}}$; -	-	$\mathcal{R}_{\texttt{src}}$;f	-	···

Fig. 17 Initiation of `map_par` with $T(\texttt{f}_i) = T_\texttt{f} = 2\texttt{p} - 4$ (stalls are noted ✗)

This is 15.93 times better than for $\texttt{p} = 1$, and therefore optimal with respect to the degree of parallelism. Fig. 18 gives the measured speedups (on the left) and execution times (on the right) for `map_par` depending on `p` and $T_\texttt{f}$ with regard to $\texttt{p} = 1$. This clearly suggests that, if the ratio $T_\texttt{f}/\texttt{p}$ is low, then the accesses dominate the computation and the speedup does not increase with `p` anymore. This can be used to deduce that, for instance, it is useless to parallelize `f` more than eight times if $T_\texttt{f} = 12$, because the resulting $\times 7.99$ speedup is optimal.

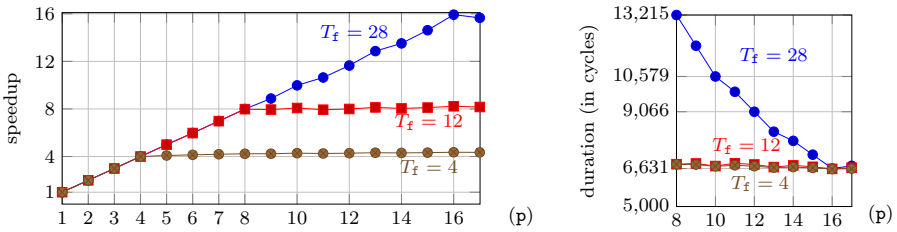


Fig. 18 Measured speedups and execution times for `map_par` in function of `p` and $T_\texttt{f}$

6 Related work

Eclat is a descendant of the Macle language [15] that has been used for hardware accelerating OCaml functions on an FPGA, in combination with a softcore-based implementation of the OCaml virtual machine. The Eclat compiler adapts the Macle compilation scheme to correctly implement the cycle-accurate semantics of Eclat. In particular, constructs *let* and $(e_1 \| e_2)$ induce no extra clock cycle in Eclat, in contrast to Macle.

Eclat is part of a long-standing tradition of Functional Hardware Description Languages. CλASH [2], for example, is subset of Haskell compiled to RTL. Computations have to be encoded as Mealy machines, which requires knowledge of both Haskell and structural hardware design.

Cement [17] extends an HDL core with control structures in imperative style while preserving cycle-accuracy and predictability. These constructs are used to explicitly synchronize pipeline stages or communicate with external modules (such as RAM blocks) responding in several cycles. The tool incorporates dynamic monitoring to detect timing violations (*e.g.*, concurrent accesses) during software-RTL simulation. Eclat pushes this approach further by providing a cycle-accurate programming language.

Synchronous languages Scade and Lustre v6 have functional arrays with predefined parallel skeletons (called *iterators*) [4]. Reads and updates (by copy)

are performed within the clock cycle. This is similar to Eclat immutable vectors. Eclat mutable arrays behave differently, due to FPGA design constraints: concurrent accesses must be sequentialized and communication with memory components induces synchronizations on clock ticks.

Whereas semantics in mainstream parallel programming (including scheduling and execution time) is usually informal, operational semantics have been proposed to model both the parallel and functional behavior of skeletons-based parallel libraries [1]. The formal semantic framework they proposed is based on a labeled transition system, which can be used to prove properties of programs. Our semantics framework is more programming-oriented. This is a similarity with BSML [10], which also has a small-step operational semantics and has been used to implement skeleton libraries. In BSML, every communication step must be followed by a synchronization step. This aims to provide language abstractions (such as higher order functions, pattern matching and exceptions) without sacrificing scalability and predictability of performance.

Reduction semantics have been used to model timing in cycle-accurate reactive languages such as Esterel [9] and ReactiveML [13]. As in our work, each cycle encompasses a sequence of small-step reductions. In these languages, given two processes p and q, reducing $(p \| q)$ can be carried $(\square \| p)$ and $(q \| \square)$. By contrast, Eclat enforces a left-to-right reduction order for $(e_1 \| e_2)$.

Recent works such as FPX [14] have tackled the difficulty of produce time and space-efficient code for FPGA architectures involving a host CPU from parallel software code. FPX is a framework to accelerate general-purpose Data Stream Processing (DSP). It provides a parametrizable Domain-Specific Language (DSL) embedded in Python, in which the application programmer can define dataflow operators as sequential programs and compose them to form the DSP application (that is a graph of dataflow operators). The programmer is no longer responsible for the low-level implementation details, which is obtained by code generation (using OpenCL as intermediate representation) combined with a runtime library implementing efficient data-flow components.

7 Conclusion

This paper has presented Eclat, an OCaml-like programming language for implementing hardware applications on FPGAs. Eclat unifies functional, parallel and synchronous programming under the global clock of the target FPGA. The semantics of the language is deterministic, by following the synchronous hypothesis [11]: all programming constructs reduce *instantaneously*, except tail-recursive function calls (which *pause* for one cycle) and memory accesses (which *pause* for two cycles, the concurrent accesses being sequentialized).

At compile time, functions defined with the "**let**" keyword are duplicated (by inlining) while those defined with "**let rec**" are shared (with dynamic dispatching of the callers for function return). Pauses on tail-calls allow the programmer to decrease the width of the combinatorial parts of the circuit.

This programming model enables performance prediction to a large extent while being quite intuitive for software programmers. It is expressive enough

to exploit fine-grained parallelism and explore the space-time trade-off inherent to FPGA programming, resulting in significant speedups with regard to sequential CPU execution. Moreover, it enables abstraction, by letting the programmer define and compose higher-order functions, including algorithmic skeletons such as *map* and *pipe*, with predictable performance.

We are currently working on enabling skeleton-based FPGA programming in a hardware implementation of the OCaml virtual machine written in Eclat [16]. As future work, we plan to provide in Eclat a Foreign Function Interface to use RTL code. We will also access external SDRAM memory from Eclat to program real-time FPGA applications with more data and scale up.

References

[1] M. Aldinucci and M. Danelutto, "Skeleton-based parallel programming: Functional and parallel semantics in a single shot," *Computer Languages, Systems & Structures (CLSS)*, vol. 33, no. 3-4, pp. 179–192, 2007.

[2] C. Baaij, M. Kooijman, J. Kuper, *et al.*, "Cλash: Structural descriptions of synchronous hardware using haskell," in *Euromicro Conference on Digital System Design (DSD): Architectures, Methods and Tools*, IEEE, 2010, pp. 714–721.

[3] R. Bakhteri, J. Cheng, and A. Semmelhack, "Design and implementation of cellular automata on FPGA for hardware acceleration," *Procedia Computer Science*, vol. 171, pp. 1999–2007, 2020.

[4] A. Benveniste, P. Caspi, S. A. Edwards, *et al.*, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, 2003.

[5] C. Bernardeschi, L. Cassano, and A. Domenici, "SRAM-based FPGA systems for safety-critical applications: A survey on design standards and proposed methodologies," *Journal of Computer Science and Tech. (JCS&T)*, vol. 30, pp. 373–390, 2015.

[6] G. Berry, "A Hardware Implementation of Pure Esterel," *Sadhana, Academy Proceedings in Engineering Sciences, Indian Academy of Science*, vol. 17, no. 1, 95–130, 1992.

[7] Y.-k. Choi, P. Zhang, P. Li, *et al.*, "HLScope+: Fast and accurate performance estimation for FPGA HLS," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, IEEE, 2017, pp. 691–698.

[8] M. Felleisen and R. Hieb, "The revised report on the syntactic theories of sequential control and state," *Theoretical computer science*, vol. 103, no. 2, pp. 235–271, 1992.

[9] S. P. Florence et al., "A calculus for Esterel: if can, can. if no can, no can.," *Proc. of the ACM on Prog. Languages (PACMPL)*, vol. 3, no. POPL, pp. 1–29, 2019.

[10] F. Gava and F. Loulergue, "Semantics of a functional BSP language with imperative features," in *Advances in Parallel Computing*, vol. 13, Elsevier, 2004, pp. 95–102.

[11] L. Gonnord, L. Henrio, L. Morel, *et al.*, "A survey on parallelism and determinism," *ACM Computing Surveys*, vol. 55, no. 10, pp. 1–28, 2023.

[12] 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.

[13] L. Mandel and M. Pouzet, "ReactiveML: a reactive extension to ML," in *Principles and practice of declarative programming (PPDP '05)*, 2005, pp. 82–93.

[14] A. Ottimo, G. Mencagli, and M. Danelutto, "Boosting general-purpose stream processing with reconfigurable hardware," *The Journal of Supercomputing*, pp. 1–31, 2024.

[15] L. Sylvestre, E. Chailloux, and J. Sérot, "Accelerating OCaml programs on FPGA," *International Journal of Parallel Prog. (IJPP)*, vol. 51, no. 2, pp. 186–207, 2023.

[16] L. Sylvestre, J. Sérot, and E. Chailloux, "Hardware implementation of OCaml using a synchronous functional language," in *Practical Aspects of Declarative Languages (PADL)*, Springer, 2024, pp. 151–168.

[17] Y. Xiao, Z. Luo, K. Zhou, *et al.*, "Cement: Streamlining FPGA hardware design with cycle-deterministic eHDL and synthesis," in *Field Programmable Gate Arrays (FPGA '24)*, 2024, pp. 211–222.