Weaving Synchronous Reactions Into the Fabric of SSA-Form Compilers

HUGO POMPOUGNAC, Inria, France
ULYSSE BEAUGNON, Google, France
ALBERT COHEN, Google, France
DUMITRU POTOP BUTUCARU, Inria, France

We investigate the programming of reactive systems combining closed-loop control with performance-intensive components such as Machine Learning (ML). Reactive control systems are often safety-critical and associated with real-time execution requirements, a domain of predilection for synchronous programming languages. Extending the high levels of assurance found in reactive control systems to computationally-intensive code remains an open issue. We tackle it by unifying concepts and algorithms from synchronous languages with abstractions commonly found in general-purpose and ML compilers. This unification across embedded and high-performance computing enables a high degree of reuse of compiler abstractions and code. We first recall commonalities between dataflow synchronous languages and the static single assignment (SSA) form of general-purpose/ML compilers. We highlight the key mechanisms of synchronous languages that SSA does not cover—denotational concepts such as synchronizing computations with an external time base, cyclic and reactive I/O, as well as the operational notions of relaxing control flow dominance and the modeling of absent values. We discover that initialization-related static analyses and code generation aspects can be fully decoupled from other aspects of synchronous semantics such as memory management and causality analysis, the latter being covered by existing dominance-based algorithms of SSA-form compilers. We show how the SSA form can be seamlessly extended to enable all SSA-based transformations and optimizations on reactive programs with synchronous concurrency. We derive a compilation flow suitable for both high-performance and reactive aspects of a control application, by embedding the Lustre dataflow synchronous language into the SSA-based MLIR/LLVM compiler infrastructure. This allows the modeling of signal processing and deep neural network inference in the (closed) loop of feedback-directed control systems. With only a minor efforts leveraging the MLIR infrastructure, the generated code matches or outperforms state-of-the-art synchronous language compilers on computationally-intensive ML applications.

1 INTRODUCTION

The Static Single Assignment (SSA) form [20, 21] has proven an extremely useful tool in the hands of compiler builders. First introduced as a representation to facilitate optimizations, it became a staple of optimizing compilers. More recently, its semantic properties—e.g., functional *determinism* while still allowing for limited *concurrency*—established it as a sound basis for High-Performance-Computing (HPC) compilation frameworks such as MLIR [17],

Authors' addresses: Hugo Pompougnac, Inria, France, hugo.pompougnac@inria.fr; Ulysse Beaugnon, Google, France, ulysse@google.com; Albert Cohen, Google, France, albertcohen@google.com; Dumitru Potop Butucaru, Inria, France, dumitru.potop@inria.fr.

XXXX-XXXX/2021/12-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

where different abstraction levels of the same application[1] share the structural and semantic principles of SSA, allowing them to co-exist while being subject to common analysis and optimization passes (in addition to specialized ones).

But while compilation frameworks such as MLIR concentrate the existing know-how in HPC compilation for virtually every execution platform, they lack a key ingredient needed in the *high-performance embedded systems* of the future—the ability to represent reactive control and real-time aspects of a system. They do not provide first-class representation and reasoning for systems with a cyclic execution model, synchronization with external time references (logical or physical), synchronization with other systems, tasks and I/O with multiple periods and execution modes.

And yet, as we shall see in this paper, while the standard SSA form does not cover these aspects, it shares strong structural and semantic ties with one of the main programming models for reactive real-time systems: *dataflow synchrony* [3, 12], and its large and structured corpus of theory and practice of reactive systems design.

Contribution. Relying on this syntactic and semantic proximity, we extend the SSA-based MLIR framework to open it to synchronous reactive programming of real-time applications. We illustrate the expressiveness of our extension through the compilation of the pure dataflow core of the Lustre language. This allows us to model and compile all data processing, computational and reactive control aspects of signal processing and machine learning applications. In the compilation of Lustre, following an initial normalization phase, all data type verifications, buffer synthesis, and causality analysis can be handled using existing MLIR SSA algorithms. Only the initialization analysis specific to the synchronous model (a.k.a. clock calculus or analysis) requires specific handling, leading to significant code reuse.

The MLIR embedding of Lustre is non-trivial. As modularity based on function calls is no longer natural due to the cyclic execution model, we introduce a *node instantiation* mechanism. We also generalize the usage of the special *undefined/absent* value in SSA semantics [9] and in low-level intermediate representations such as LLVM IR [15]. We clarify its semantics and relate it to the notion of absence and the associated static analyses (*clock calculi*) of synchronous languages [2].

Our extension remains fully compatible with SSA analysis and code transformation algorithms. It allows giving semantics and an implementation to all correct SSA specifications. It also supports static analyses determining correctness from a synchronous semantics point of view.

Outline. Section 2 formalizes the SSA semantics. Section 3 extends SSA to allow reactive synchronous programming. Section 4 covers the embedding of Lustre into MLIR SSA and its compilation. In Section 5 we present our use cases. We discuss related work in Section 6 before the conclusion in Section 7.

2 SSA SYNTAX AND SEMANTICS

The syntax and semantics of SSA form have been presented multiple times before, but we need to settle on one as a basis for extension. The SSA variant we selected is based on MLIR [17], which uses continuation-passing style (CPS) branch arguments rather than ϕ operators.

[1]Ranging from ML dataflow graphs and linear algebra specifications down to affine loop nests and optimized (tiled, vectorized...) low-level code.

```
<ssa_spec> ::= <function>+
<function> ::= func <fun_name> <fun_iface> <fun_body>
<fun_iface> ::= (<type>*)->(<type>*)
<fun_body> ::= <block>+
<block> ::= <blk_arg>:<blk_body>
<blk_arg> ::= <block_name>(<tvar>*)
<tvar> ::= <var>:<type>
<blk_body> ::= <op>* <term_op>
<op> ::= (<var>*)=<op_id>(<tvar>*):<type>*
       | <var> = load(<tvar>):<type>
       | store(<tvar>,<tvar>)
<term_op> ::= cond_br <var> <blk_arg> <blk_arg>
            | br <blk_arg> | return(<tvar>*)
<op_id> ::= <arith_op> | <bool_op> | call <fun_name>
```

Fig. 1. SSA syntax

Fig. 1 presents a minimal SSA syntax (in black) extended with the constructs for functional composition (in red), and with the `load` and `store` operations to represent of memory side effects (in green).[2]

Additional structural properties must be met. No two blocks of a function may have the same name, and branching operations (`br` and `cond_br`) may only reference existing blocks with the correct number of arguments. The blocks and the branching operations of an SSA function determine a sequential control flow graph (SCFG) structure, with the first block serving as control entry point. Execution proceeds sequentially inside each block. Each variable is either defined from one operation, or the argument of a single block header.[3] No two functions may have the same name, and calling a function assumes that it exists and has the correct interface. The interface of a function consists in the arguments of its first block which are also the function arguments, and the arguments of its `return` operation which are the function return values. If multiple `return` operations exist in a function, they must have the same number of arguments. All SSA blocks are terminated by a `return` or a branch operation. These operations are called *terminators*.

The small example of Fig. 2 illustrates the specific properties of reactive systems and the differences between Lustre and SSA. The Lustre node (left) and MLIR SSA function (right) implement the same functionality, but we focus on the SSA function in this section. As we will see later, the latter could be the result of the compilation of the former. From now on, and to simplify the exposition, we use the input/output terminology of Lustre nodes for MLIR block and function arguments/return values. We will see that this terminology is well supported by semantics properties. Note that the MLIR SSA syntax requires that block identifiers start with "^" and that all variable names start with "%". MLIR also provides more intuitive forms for specific operations: for example, function calls specify the full function signature at their end; the comparison taking two input variables as arguments and producing one variable (the test result) has the syntax of line 10, which specifies the kind

[2]The syntax of types is left implicit. The following semantics is largely independent on the type system for SSA expressions.

[3]Under classical SSA notation, this amounts to the variable being defined from exactly one ϕ operator. In classical SSA notation, ϕ operators are used in blocks that are destination of multiple branching operations. Each one allows the construction of a unique value based on values coming from the different source blocks.

```
1  node stepper_drv(inc:int) returns (pos:int)
2  var pos_pre,pos_inc,pos_tmp,cst:int; ck:bool;
3  let
4
5
6    pos_pre = 0 fby pos;
7
8    pos_inc = pos_pre+inc;
9    pos_tmp = pos_inc-10;
10   ck = (pos_tmp >= 0);
11   pos = if ck then pos_tmp else pos_inc;
12
13   cst = 0 when ck ;
14   actuate(cst);
15
16
17
18
19
20 tel
```

```
1  func @stepper_drv()->() {
2  ^start:
3    %c0 = constant 0 : i32
4    %c10 = constant 10 : i32
5    br ^step(%c0:i32)
6  ^step(%pos_pre:i32):
7    %inc = call @input_inc() : () -> (i32)
8    %pos_inc = addi %pos_pre, %inc : i32
9    %pos_tmp = subi %pos_inc, %c10 : i32
10   %ck = cmpi "sge", %pos_tmp, %c0 : i32
11   %pos = select %ck,%pos_tmp,%pos_inc : i32
12   cond_br %ck, ^act(%c0:i32), ^out
13 ^act(%cst:i32):
14   call @actuate(%cst) : (i32) -> ()
15   br ^out
16 ^out:
17   call @output_pos(%pos) : (i32)->()
18   call @tick() : () -> ()
19   br ^step(%pos:i32)
20 }
```

Fig. 2. Stepper motor driver in Lustre (left) and MLIR SSA (right). Control statements in green, data processing operations in red, cyclic I/O and time synchronization in violet, state manipulation in blue.

of test (`sge` for `>=`), the type of input data (`i32`), but not the type of the Boolean output which is implicit (`i1`); operation `select` in line 11 outputs one of its data inputs based on the value of its Boolean test variable `%ck` (of type `i1`).

Our example is a driver for a stepper motor which receives rotation commands as increments of 0.18°, but can only actuate—issue physical commands to the motor—for 1.8° at a time. For this reason, commands must be buffered and only actuated when their number exceeds 10. This behavior, typical of an embedded control system, involves a continuous interaction with the environment. In our example, this interaction is driven by the infinite loop formed by the SSA branching operations. Each iteration of the loop is an *execution cycle* during which the program reads its input `%inc` from its environment, performs computations, potentially actuates the motor and finally outputs `%pos`. Timely interaction with the environment is achieved through the function calls in violet, which are discussed in Section 3.

2.1 SSA operational semantics

For conciseness, we shall assume all variables (including Booleans) are semantically represented as integers. We also assume each operation of the program has a unique label, such as a program line number if we assume that no two operations share the same line.

Notations. The cardinal of a set $\mathcal{S}$ is denoted $|\mathcal{S}|$. We use the OCaml notation for lists: [] is the empty list, $h :: t$ the list of first element h and tail t.

Int is the domain of the integer type. To represent the status of a memory location or variable that has not been initialized, we use the special *undefined* value denoted $\perp$. Therefore, the status of any variable or memory location, at any execution point, is an element of $\overline{Int} = Int \cup \{\perp\}$. We also denote $\perp$ any function that always returns $\perp$, regardless of its domain. Given a (mathematical) function f and a value x of its domain, $f[x \leftarrow y]$ is the function that is identical with f everywhere except on x, where it has value y.

$\mathcal{L}^{\mathtt{f}}$ and $\mathcal{V}^{\mathtt{f}}$ are respectively the sets of labels and of variables of an SSA function `f`, and $b_0^{\mathtt{f}}$ is its first block. The function containing block b is denoted $fun(b)$. The ordered set of inputs of block b is $in(b) \subseteq \mathcal{V}^{fun(\mathtt{b})}$, and $in(b)_i$ is the i^{th} input of b. The ordered set of variables

$$(Start^{\mathtt{f}}(v_1,\ldots,v_{|in(b_0^{\mathtt{f}})|}),cs,m) \to (Run^{\mathtt{f}}(fst(b_0^{\mathtt{f}}),\bot[in(b_0^{\mathtt{f}})_k \leftarrow v_k \mid 1 \le k \le |in(b_0^{\mathtt{f}})|]),cs,mem) \quad \textbf{(start)}$$

$$\frac{op(l) = \text{“}(v_1,\ldots,v_n) = op\_id(w_1,\ldots,w_m)\text{“} \qquad (o_1,\ldots,o_n) = [\![op\_id]\!](s(w_1)\ldots,s(w_m))}{(Run^{\mathtt{f}}(l,s),cs,m) \to (Run^{\mathtt{f}}(nxt(l),s[v_i \leftarrow o_i \mid 1 \le i \le n]),cs,m)} \quad \textbf{(opcall)}$$

$$\frac{op(l) = \text{“}\mathtt{br}\ bb(v_1,\ldots,v_{|in(bb)|})\text{“}}{\begin{array}{l}(Run^{\mathtt{f}}(l,s),cs,m) \to \\ \quad (Run^{\mathtt{f}}(fst(bb),s[in(bb)_i \leftarrow s(v_i) \mid 1 \le i \le |in(bb)|][loc(bb)_i \leftarrow \bot \mid 1 \le i \le |loc(bb)|]),cs,m)\end{array}} \quad \textbf{(goto)}$$

$$\frac{op(l) = \text{“}\mathtt{cond\_br}\ v\ bb1(v_1,\ldots,v_{|in(bb1)|})\ bb2(w_1,\ldots,w_{|in(bb2)|})\text{“} \qquad s(v) \notin \{0,\bot\}}{\begin{array}{l}(Run^{\mathtt{f}}(l,s),cs,m) \to \\ \quad (Run^{\mathtt{f}}(fst(bb1),s[in(bb1)_i \leftarrow s(v_i) \mid 1 \le i \le |in(bb1)|][loc(bb1)_i \leftarrow \bot \mid 1 \le i \le |loc(bb1)|]),cs,m)\end{array}} \quad \textbf{(ifthen)}$$

$$\frac{op(l) = \text{“}\mathtt{cond\_br}\ v\ bb1(v_1,\ldots,v_{|in(bb1)|})\ bb2(w_1,\ldots,w_{|in(bb2)|})\text{“} \qquad s(v) = 0}{\begin{array}{l}(Run^{\mathtt{f}}(l,s),cs,m) \to \\ \quad (Run^{\mathtt{f}}(fst(bb2),s[in(bb2)_i \leftarrow s(w_i) \mid 1 \le i \le |in(bb2)|][loc(bb2)_i \leftarrow \bot \mid 1 \le i \le |loc(bb2)|]),cs,m)\end{array}} \quad \textbf{(ifelse)}$$

$$\frac{op(l) = \text{“}\mathtt{return}(v_1,\ldots,v_{Of})\text{“}}{(Run^{\mathtt{f}}(l,s),cs,m) \to (End^{\mathtt{f}}(s(v_1),\ldots,s(v_{Of})),cs,m)} \quad \textbf{(end)}$$

$$\frac{op(l) = \text{“}(v_1,\ldots,v_n) = \mathtt{call}\ \mathtt{g}(w_1,\ldots,w_k)\text{“}}{(Run^{\mathtt{f}}(l,s),cs,m) \to (Start^{\mathtt{g}}(s(w_1),\ldots,s(w_k)),Run^{\mathtt{f}}(l,s)::cs,m)} \quad \textbf{(call)}$$

$$\frac{op(l) = \text{“}(v_1,\ldots,v_n) = \mathtt{call}\ \mathtt{g}(w_1,\ldots,w_k)\text{“}}{(End^{\mathtt{g}}(x_1,\ldots,x_n),Run^{\mathtt{f}}(l,s)::cs,m) \to (Run^{\mathtt{f}}(nxt(l),s[v_i \leftarrow x_i \mid 1 \le i \le n]),cs,m)} \quad \textbf{(ret)}$$

$$\frac{op(l) = \text{“}w = \mathtt{load}(addr)\text{“} \qquad s(addr) \neq \bot \qquad valid\_addr(s(addr))}{(Run^{\mathtt{f}}(l,s),cs,m) \to (Run^{\mathtt{f}}(nxt(l),s[w \leftarrow m(s(addr))]),cs,m)} \quad \textbf{(load)}$$

$$\frac{op(l) = \text{“}\mathtt{store}(addr,v)\text{“} \qquad s(addr) \neq \bot \qquad valid\_addr(s(addr))}{(Run^{\mathtt{f}}(l,s),cs,m) \to (Run^{\mathtt{f}}(nxt(l),s),cs,m[s(addr) \leftarrow s(v)])} \quad \textbf{(store)}$$

Fig. 3. SSA semantics. Memory access rules in green. Function call rules in red.

assigned by operations of a block b is denoted $loc(b)$. The label of the first operation in block b is $fst(b)$. The operation associated with a label $l \in \mathcal{L}^{\mathtt{f}}$ is denoted $op(l)$. If $op(l)$ is not a terminator, then $nxt(l)$ is the label of the next operation in its block. The number of arguments of a `return` operation of function `f` is denoted O^f.

Execution state. The execution state of an SSA *function* `f` is one of:

- An initial state $Start^{\mathtt{f}}(v_1,\ldots,v_{|in(b_0^{\mathtt{f}})|})$, where $v_i \in \overline{Int}$ are the actual parameters of the function.
- A final state $End^{\mathtt{f}}(w_1,\ldots,w_{Of})$, where $w_i \in \overline{Int}$ are the outputs of `f` (received as input by `return`).
- A triple $Run^{\mathtt{f}}(pc,val)$ formed of the label pc of the operation to execute next (the program counter) and a partial valuation $val : \mathcal{V}^{\mathtt{f}} \to \overline{Int}$ of all the variables.

The execution state of an SSA *specification* is a triple (s,cs,m) formed of the state s of the function that is currently executing, a list cs of running states representing the call stack, and the current state $m : Int \to \overline{Int}$ of the memory. An *initial state* of the specification has the form $(Start^{\mathtt{f}}(\ldots),[],m)$, where m is the initial memory state and `f` the function that serves as execution entry point. A *final state* of the specification has the form $(End^{\mathtt{f}}(\ldots),[],m)$.

Program execution. Transition rules are provided in Fig. 3. An *execution trace* of an SSA specification is any sequence of transitions starting in an initial state. Note that if $t = s_0 \to s_1 \to \ldots \to s_n \to \ldots$ is a trace, then any prefix $t_n = s_0 \to s_1 \to \ldots \to s_n$ is also a trace. We denote with $\le$ the prefix order on traces, meaning we can write $t_n \le t$.

Note that rules **(ifthen)** and **(ifelse)** define the behavior of a conditional branch only when the test variable is defined. When it is $\perp$, execution cannot advance—it blocks. Execution also blocks when a `load` or `store` operation accesses an address which is either $\perp$ or invalid. Under the LLVM interpretation of undefinedness [15], this amounts to assuming that the use of $\perp$ in decisions and computations is an *immediate undefined behavior*. To perfectly match this interpretation, we also assume that the semantics $[\![op\_id]\!]$ of each operation is a partial function, undefined when inputs have value $\perp$.

Note that maximal traces (in the sense of $\leq$) will either be infinite (when execution never ends) or end in a final state, or end when execution blocks.

Separation assumptions. Our semantics separates, in both state expression and transition rules, the part corresponding to the SSA core from the extensions needed to represent function calls and memory. For instance, to consider only the core SSA semantics one simply has to remove call stack and memory terms from the specification state, and only consider the black rules of Fig. 3 (which do not access memory or call stack). Similarly, function call rules do not access the memory, and memory rules do not access the call stack.

2.2 Determinism and correctness

If we assume that all operations are deterministic (which in our formal framework amounts to assuming that $[\![op\_id]\!]$ are partial functions) the sequentiality of the SSA execution implies its determinism: for each initial state s there exists a unique trace starting in s that is maximal in the sense of $\leq$.

But determinism is not sufficient. Correctness also requires that execution never blocks. To ensure this, SSA enforces a stronger property: that all variables are different from $\perp$ whenever used in decisions, memory access adresses, and computations. Assume that variable v is defined by block b, either by an operation of b or by the block header itself. Also assume that v is an input to operation o' of block b'. To ensure that v does not have value $\perp$ when o' is executed, one must ensure (as a necessary property) that any execution path reaching o' passes through the definition of v. This happens *if and only if* one of the following conditions is true:

D1 $b = b'$, and the definition of v precedes o' in b, either as block argument, or as output of an operation.

D2 $b \neq b'$, and any possible execution reaching b' necessarily passes through b.

While checking property **D1** is straightforward, determining that **D2** holds for any variable v and operation o using v is not tractable in the general case (Boolean satisfiability can be reduced to the decision of **D2**).

For this reason, SSA-based compilation always ensures **D2** by enforcing a sufficient property, named dominance, which can be checked using a low-complexity structural analysis of the SSA SCFG [8]. Dominance is considered part of SSA form correctness, along with the syntactic correctness and the structural properties of Section 2.

Together, these structural properties ensure the correctness of SSA specifications that do not access memory. When memory is used, these properties must be complemented by a proof of the fact that each memory location is initialized before it is read.

3 FROM SSA TO SYNCHRONOUS CONCURRENCY

In this section we extend the syntax and semantics of SSA with the operational mechanisms needed to represent synchronous concurrency. The extension leaves the (non-reactive) SSA semantics of Fig. 3 unchanged, and allows the application of all SSA code transformations.

```
<type> += in(<type>) | out(<type>)
 <op> += (<var>?) = tick(<tvar>*)
       | <var> = sync(<tvar> <tvar>+)
       | <var> = input(<var>):<type>
       | (<var>?) = output(<var>:<var>):<type>
       | <var> = undef:<type>
<op_id> += inst <fun_name> <inst_id>
```

Fig. 4. SSA syntax extensions for synchronous reactive programming. "+=" extends an already-defined non-terminal. Grayed non-terminals are those of Fig. 1.

Thus, building on MLIR's extensible syntax and semantics, it is possible to model and generate code for reactive real-time applications without changing the existing code base (only introducing additional behaviors).

3.1 Cyclic execution

To represent the behavior of embedded systems, which interact with their environment in a continual and timely fashion, all synchronous languages have *cyclic execution models*. The execution of a synchronous program is an *a priori* infinite *sequence of execution cycles*. Execution cycles being non-overlapping, they form a *logical time base*: each operation happens in exactly one logical cycle which can be identified by its index. But synchronous logical time is not only a descriptive notion used during analysis. It is meant to allow the synchronization of cycle execution onto external time bases. For instance, periodic real-time execution is typically enforced by synchronizing cycle triggering (the logical time base) with a periodic HW timer.

The SSA form also allows the representation of cyclic behaviors under the form of cyclic CFGs, as exemplified in Fig. 2. However, what constitutes an *execution cycle* is not clearly identified, nor the mechanisms for synchronizing execution cycles with an external time base, nor how to constrain an operation to happen in a specific cycle. To allow SSA-based embedded real-time programming, we must allow the specification of all these aspects *in a way that will be preserved by SSA-based code transformations and optimizations*.

Tying the definition of execution cycles to the structural elements of SSA[4] is tempting, but unfeasible, as basic blocks are not preserved by common optimizations. Another tempting approach is to extend the syntax *and semantics* of basic blocks and/or branching statements to explicitly identify some of them as execution cycle barriers. However, such an approach would require changes to much of existing SSA-related compiler code (such as dominance analysis or optimization algorithms), which we want to avoid.

3.1.1 The `tick` operation. The solution we chose is the introduction of a new operation `tick` to identify execution cycle barriers. To order operations with respect to these barriers, `tick` relies on two SSA mechanisms: dominance and memory access ordering.

Dominance-based ordering. The syntax of `tick`, provided in Fig. 4, shows that it can take as arguments any number of variables, which allows specifying that operations producing these variables are executed before the cycle barrier. It produces an optional variable `%s` of type `none`, which is the unit type of MLIR SSA.

[4]For instance, by annotating specific basic blocks or specific branching operations to identify them as cycle transitions.

```
1  func @periodic1()->() {
2  ^reset:
3
4    %x0 = call @init():()->tensor<64xi8>
5    br ^step(%x0:tensor<64xi8>)
6  ^step(%x1:tensor<64xi8>):
7    %x2 = addi %x1, %x1: tensor<64xi8>
8    %s = tick(%x2:tensor<64xi8>)
9    %x3 = sync(%x2:tensor<64xi8>,%s:none)
10   br ^step(%x3:tensor<64xi8>)
11 }
```

```
1  func @periodic2() -> () {
2  ^reset:
3    %x0 = alloc(): memref<64xi8>
4    call @init(%x0): :(memref<64xi8>)->()
5    br ^step
6  ^step:
7    call @do_addi(%x0) :(memref<64xi8>)->()
8    tick()
9
10   br ^step
11 }
```

Fig. 5. Synchronizing computations w.r.t. execution cycle barriers (`tick`) to enforce periodic execution. Dominance-based (left) vs. side-effect-based (right).

Like for the unit type of functional programming or the *pure signals* of synchronous programming [19], a variable of type `none` represents *pure synchronization*. It carries no information but requires operations reading it to happen after the operation producing it. To facilitate the specification of such ordering constraints, we also introduce the `sync` operation which allows transferring the dependences of one or more variables onto another. The operation takes as input two or more variables. It copies the value of its first input onto its output as soon as all inputs have arrived.

As Fig. 5(left) shows, this way of enforcing ordering is particularly useful early in the compilation process, when aggregate n-dimensional data are represented with abstract types (such as the *tensors* of MLIR) manipulated with side-effect-free operations.

Ordering by side-effects. We also assume that `tick` has unspecified side-effects, which means it cannot commute (during SSA code transformations) with operations that read or write memory. This way of enforcing ordering is particularly useful later in the code generation process, once buffer allocation of aggregate data has been done. This is the case in Fig. 5(right), which could be an implementation of the program in Fig. 5(left). Note that in both cases SSA code transformations (e.g., loop unrolling) can be applied freely, without changing the ordering of computations w.r.t. execution cycle barriers.

Structural properties. To ensure that the execution of a *reactive SSA function* is an infinite sequence of execution cycles, we will require that it does not contain `return` operations, and that each potentially unbounded cycle of the SSA specification contains at least one `tick` operation.

3.1.2 Compilation. The transformation of a standard SSA form specification into executable sequential code is a well-understood process. However, the introduction of `tick` fundamentally changes the SSA semantics by requiring a cyclic interaction with the environment/scheduler.

The way this interaction is traditionally implemented in the compilation of synchronous languages [3, 4, 19] is illustrated in Fig. 6. The control flow of the source code (function `@n` in our case) is completely restructured (inverted) in order to produce a semantically equivalent program with a single `tick` operation (function `@n_drv`). To do this, the control and data state of the original function *between execution cycles* must be explicitly represented. In our example, this state is transmitted by the arguments of `^step` and consists of:

- A Boolean value (of type `i1`) to determine whether `@f` or `@g` should execute in the next cycle.
- A value of type `i32` allowing the transmission of the output of `@f` to `@g`.

```
func @n()->() {
^start:
  %x = call @f():()->(i32)
  tick()
  call @g(%x):(i32)->()
  tick()
  br ^start
}
```

```
func @n_step(i1,i32)->(i1,i32){
^start(%s:i1,%x:i32):
  cond_br %s, ^true,^false
^true:
  %x1 = call @f():()->(i32)
  %false = constant 0:i1
  return %false,%x1:i1,i32
^false:
  call @g(%x):(i32)->()
  %true = constant 1:i1
  return %true,%x:i1,i32
}
```

```
func @n_drv()->() {
^start:
  %s0 = constant 0 : i1
  %x0 = constant 0 : i32
  br ^step(%s0,%x0:i1,i32)
^step(%s:i1,%x:i32):
  %s1,%x1 = call @n_step(%s,%x)
      :(i1,i32)->(i1,i32)
  tick()
  br ^step(%s1,%x1:i1,i32)
}
```

Fig. 6. Synchronous languages approach to code generation. Source code left, output middle and right.

At the beginning of each cycle the state is decoded, the needed computations are triggered, then a new state is encoded and transmitted to the next cycle. This process is usually represented with a separate *step function*, in our case `@n_step`.

Classical synchronous language compilers will usually produce just this step function and the data structure describing the application state. When used in conjunction with the dataflow modularity described in Section 3.3, this approach allows modular code generation[5] [4]. However, the driver function (in our case `@n_drv`), and in particular the implementation of `tick`, are usually not generated, being considered too implementation-dependent.

This compilation approach has been long tested in practice [2, 12], where it has shown its strengths (most notably modularity), but also its limits. The limits are mainly related to the one-size-fits-all generated code with a single step function and a state representation that must cover the needs of *all* execution cycle transitions. In our example, in the compilation output, the state variable of type `i32` is assigned a value and then passed on to the next cycle *in every cycle*, even though it is semantically produced (output of `@f`) only in odd cycles. A first evaluation of the inefficiency of the inverted state representation is provided in previous work on state representation optimization for the Esterel language [19].[6] It is important to note that, once the generation of the step function performed, classical compiler optimizations are confined to the scope of the step function, and optimizations involving multiple execution cycles must be specifically designed for each particular language and state encoding.

Our compilation method cannot follow this example and systematically restructure control to obtain a loop with a single `tick`. Not only because of the potential efficiency loss, but because in many cases *the implementation must have a different structure*. For instance, in avionics MIF/MAF applications [10] the implementation must have a structure similar to that of Fig. 6(left), where a global periodic pattern (the global loop, named *major frame*, or MAF) is split by the `tick` operations into time intervals of equal length (the *minor frames*, or MIFs), each containing a different code.

To allow the implementation, without restructuring, of any reactive SSA graph satisfying the structural properties defined above, we propose a return to the fundamentals of reactive systems design, by making explicit the interaction with the system scheduler. In our compilation approach, a reactive SSA specification such as function `@n` of Fig. 6(left) is seen as

[5]One step function per hierarchic synchronous module, the state representation of a module including that of sub-modules it hierarchically includes, and its step function calling the step functions of sub-modules.
[6]Memory allocation for synchronous languages also poses other challenges, such as the fact that modular code generation requires in the general case an inefficient encoding for memory constructs, which uses not one, but two variables for each state element. Several papers address this issue in previous work [10, 11].

a sequential process running under a cooperative multi-tasking scheduler. Each time the execution of `@n` reaches a `tick` operation, the execution context (state) of `@n` is saved and control is given back to the scheduler. When the scheduler determines that a new execution cycle must be triggered, it restores the state of the process, execution of the `tick` operation terminates, and the scheduler hands returns control to the process.

This operational mechanism, whose formal semantics is defined in Section 3.5, can be easily implemented on various execution platforms ranging from low-level timers on bare metal platforms [10] to POSIX system services such as `longjmp` (as we do in Section 5) to coroutine mechanisms of various languages and to RTOS services such as `PERIODIC_WAIT` in the avionics-oriented IMA/ARINC 653 standard [1].

Note that our proposal does not exclude the classical approach of compiling synchronous languages. Instead, it is complementary, allowing the modeling of implementation aspects that were previously not covered by code generation, and by allowing more expressiveness in the implementation.

3.2 Cyclic I/O

An embedded system will continually interact with its environment, cyclically reading inputs and writing outputs. In practical implementations, this is typically done by reading and writing memory-mapped registers that can be represented with volatile C variables, or by calling dedicated I/O functions. Synchronous languages abstract away such implementation-dependent mechanisms under the form of *input and output variables* that can be read or written at each cycle, with two constraints related to the synchronous model:

- An output variable can be written at most once per execution cycle.
- All reads of an input variable during an execution cycle must return the same result.

For instance, in Fig. 2, the input `inc` of the Lustre program is read at each execution cycle.

By comparison, when not considering memory side effects, MLIR functions interact with their environment only twice:

- At the beginning of their execution, to read the value of their input arguments, which then remains constant during the execution of the function.
- When reaching a `return` operation, when the function completes.

These assumptions enable common SSA optimizations such as loop-invariant code motion. Volatile accesses to memory locations can sometimes be represented, like in the low level dialects of MLIR (LLVM, SPIRV), but they represent a particular low-level implementation, excluding others.

The solution we chose to represent cyclic I/O is based on the representation of input and output *channels* with function arguments of the special types `in(t)` and `out(t)`, where `t` is the type of data transmitted by the channel. Access to channel variables is done exclusively using the `input` and `output` operations, whose syntax is provided in Fig. 4. Operation `input` has a single argument of input channel type. Each time it is executed, `input` samples the channel for a value of the correct type, which is placed in the output variable. An `output` operation has two inputs: one output channel and a second variable (of the corresponding non-channel type) whose value will be written to the output.

To specify ordering relations between I/O operations and other operations, we use the same mechanisms discussed for `tick` in Section 3.1.1. On both `input` and `output` we assume unspecified side-effects, which prevents reordering with other operations that access memory and function calls. Both I/O operations also allow dominance-based ordering. For this reason, `output` has an (optional) output of type `none`.

```
1 func @n(in(i32), out(i32))->() {
2 ^start(%i:in(i32), %o:out(i32)):
3   br ^step
4 ^step:
5   %x = input(%i):i32
6   output(%o:%x):i32
7   tick()
8   br ^step
9 }
```

```
1 func @n(()->(i32), (i32)->())->() {
2 ^start(%i:()->(i32), %o:(i32)->()):
3   br ^step
4 ^step:
5   %x = call_indirect %i (): ()->(i32)
6   call_indirect %o (%x): (i32)->()
7   call @tick() : ()->()
8   br ^step
9 }
```

Fig. 7. Code generation for I/O operations

Structural properties. As a limitation specific to our current implementation, it is required that only functions representing reactive behaviors use channel variables or the operations `input` and `output`. It is also required that reactive functions have only arguments of I/O channel type, and that no channel variable is output of an operation.

To comply with the synchronous model requirement that each variable has at most one value during each cycle (we will further develop this aspect in Section **??**) it is also required that at most one `input` or `output` operation is performed on a given I/O variable between two instances of `tick`. This semantic property is usually enforced by requiring the respect of structural properties, e.g., by requiring that every CFG path between two operations on the same channel contains a `tick` operation.

Compilation. As explained above, various low-level mechanisms can be used to implement the operations `input` and `output`, the most typical being volatile variables and calls to I/O functions. For portability, our compiler takes the second approach, by:

- Transforming each function argument of type `in(t)` into a function argument of type `()->(t)`, i.e., a reference to a function that takes no argument and produces one result of type `t`.
- Transforming each function argument of type `out(t)` into a function argument of type `(t)->()`.
- Each `input` and `output` operation is transformed into a call to the corresponding function argument. Note the use of `call_indirect`, a version of operation `call` allowing calling a function transmitted by reference.

Fig. 7 provides a small example: a reactive program with one input channel and one output channel that copies at each cycle its input on the output.

When the value produced by `input` or taken as input by `output` is an object stored in memory,[7] much care must be exerted to avoid memory errors (accessing unallocated memory and memory leaks). In our implementation, we will assume that all memory-stored objects produced by an `input` operation are allocated and deallocated by the environment, with a lifetime finishing at the end of the current execution cycle. We also assume that memory-stored objects given as argument to an `output` operation are allocated and deallocated by the function, and that the environment no longer uses them when the next execution cycle begins.

3.3 Modularity

The modularity of SSA is that of sequential procedural programming. The modules of an SSA specification are the functions, which interact through *function calls*.

[7]Like the variables of type `memref` in MLIR.

```
func @main(in(i32),out(i32))->(){
^start(%ic:in(i32),%oc:out(i32)):
  br ^step
^step:
  %i = input(%ic):i32
  %x1 = inst @sum "a" (%i:i32)
  tick()
  %x2 = inst @sum "a" (%i:i32)
  output(%oc:%x2):i32
  tick()
  br ^step
}
```

```
func @sum(in(i32),out(i32))->(){
^start(%ic:in(i32),%oc:out(i32)):
  %s0 = constant 0 : i32
  br ^step(%s0:i32)
^step(%s:i32):
  %i = input(%ic):i32
  %s1 = addi %s, %i : i32
  tick(%s1:i32)
  %o = call @f(%s1):(i32)->(i32)
  output(%oc:%o) : i32
  tick()
  br ^step(%s1:i32)
}
```

Fig. 8. Submodule instantiation example

By comparison, the formal models underpinning synchronous languages are concurrent. In the most general settings, such as Esterel's constructive semantics [19], *the execution of two sub-modules of a specification can advance concurrently, synchronizing and communicating with each other in both ways.* Determining that the execution of such a specification does not block (a process known as *causality analysis*) is in general undecidable, if integer data are allowed, and NP-hard (untractable in practice) if the input language uses only Boolean variables. Furthermore, the implementation of such general specifications can be very inefficient due to intricate semantic rules.

For this reason, synchronous language compilers have early on imposed simple structural constraints allowing fast and modular compilation: in each module, the computations of an execution cycle must form an acyclic dependency graph, allowing fast scheduling and code generation. In this acyclic graph, to allow the separate compilation of a sub-module, its computations must be grouped together as a single graph node, meaning that they can be performed *atomically.*

To allow the representation of this mechanism in our SSA extension, we introduce the notion of *instance* of a reactive function `@f`, which is a process (with separate state) executing function `%f` under the system scheduler. Instances are uniquely identified (with lists of strings, in our implementation). We assume that the first reactive function of a specification has an instance identified with the empty list of strings `[]` that receives control when the system starts. All other instances are inductively defined and possibly given control during execution using the operation `inst` (syntax in Fig. 4): if `i` is an instance of function `@f` which contains operation "`inst @g str`", then instance `str::i` of function `@g` is automatically defined.

We provide in Fig. 8 an example of submodule instantiation. The system has two reactive functions and two instances: the implicit `[]` instance of function `@main` and one instance `["a"]` of function `@sum`. Notice that the two `inst` operations of `@sum` trigger execution cycles of the same instance (`["a"]`).[8] Instance `[]` reads its unique input from the environment on odd cycles, triggers one tick of instance `["a"]` in every cycle (giving it as input the value of `%i`) and outputs the last output of `["a"]` in even cycles. Instance `["a"]` of `@sum` has a state in which it accumulates the sum of inputs it receives on odd cycles. On even cycles, it computes and outputs variable `%o`.

Structural constraints. The number and types of input and output variables of an `inst` operation must match the signature of the reactive function it instantiates.

[8]In the presence of periodic activation patterns, this enables the unrolling of the infinite loop over the period, followed by constant propagation, which eliminates activation tests over the iteration counter.

Compilation. The compilation method based on conversion to step functions will implement sub-module instantiation using function calls [4]. When conversion to a step function is not desired, we transform each instance into a process running under a cooperative scheduler. This mechanism extends that of Section 3.1.2 by clarifying how the scheduler passes the control between instances. When reaching an `inst` operation, the inputs and the control are transmitted to the instance, which then executes until reaching a `tick` operation, at which point it saves its state and returns the outputs to the caller, according to the rules of Section 3.5.

3.4 Signal absence

Consider the simple example in Fig. 6 (left) and its implementation on the right. Note that the translation slightly changes the internal behavior of the program: on the left, the variable `%x` is transmitted only from odd cycles to even cycles. But in the translation result `%x` has been added to the program state, which is computed and transmitted at every cycle. Thus, in the `^false:` basic block of `@n_step`, under SSA semantics, we need to return a value for `%x` at each cycle, even if the source program does not require it.

In this case, we made the natural choice of maintaining the previous value, which later allows the encoding of the state in (persistent) memory. But using instead a constant of type `i32` or a dynamically-computed value would also have been correct, as this value is never used in computations. The situation is similar in Fig. 8: function `@sum` outputs values only on even cycles, meaning that the value of `%x1` in function `@main` is never correctly initialized, yet the specification is overall deterministic, because this *absent/undefined* value is never used in computations.

Such situations are common in the synchronous modeling of multi-rate and multi-periodic systems, which explains why *absence* prominently figures in the semantics of all synchronous languages [2]. As we shall see in Section 4, in the Lustre language a variable that is *absent* in a synchronous execution cycle has an *undefined* value and cannot be used in decisions or computations. This definition fully matches that of $\perp$ in Section 2.1.

However, synchronous languages such as Lustre are designed to support low-complexity analyses (known as *clock calculi*) ensuring that no uninitialized data is ever used in computations (much like the SSA form is designed to support dominance analysis). These analyses exploit the tight coupling between (dataflow) control propagation and initialization, as well as stringent synchronization requirements (known as *clock constraints*) enforced by each language primitive.

But our reactive extension of SSA is meant to be an intermediate compiler representation, where efficient encodings may require the *explicit representation and manipulation of absence/undefinedness*, which breaks the coupling between SSA control propagation and initialization. This situation is similar to that of LLVM IR, which extends the basic SSA syntax and semantics with not one, but two types of undefinedness to support different analysis and optimization approaches—`llvm.undef` and `llvm.poison` [15].

$$\frac{op(l) = \text{“}v = \texttt{undef}\text{“}}{(Run^{\mathrm{f}}(l,s),cs,m) \rightarrow (Run^{\mathrm{f}}(nxt(l),s[v \leftarrow \perp]),cs,m)}\ \textbf{(undef)}$$

Fig. 9. Semantics of `undef`

Unlike LLVM IR, we do not need to add a new semantic value. Instead, we simply allow the explicit manipulation of $\perp$ without breaking the structural correctness rules of SSA. To do this, we introduce operation `undef` (syntax in Fig. 4, semantics in Fig. 9). This

operation sets its output variable to $\perp$. To understand how this value is propagated, consider function `@n_step` of Fig. 6, and recall that in this function, in the `^false` basic block, the second argument of `return` could take any value instead of `%x`, because it is never used in computations. Using operation `undef`, we can rewrite this function to avoid the need to return a defined value, as pictured in Fig. 10. This not only faithfully represents the original behavior of Fig. 6 (left), but also allows more efficient code generation, as detailed below.

```
func @n_step(i1,i32)->(i1,i32){
^start(%s:i1,%x:i32):
  cond_br %s, ^true,^false
^true:
  %x1 = call @f():()->(i32)
  %false = constant 0:i1
  return %false,%x1:i1,i32
^false:
  call @g(%x):(i32)->()
  %ux = undef(): i32
  %true = constant 1:i1
  return %true,%ux:i1,i32
}
```

Fig. 10. Use of `undef` (in blue) to represent signal absence

Compilation of `undef`. Mastering undefinedness in LLVM is notoriously difficult [15]. However, this difficulty is a consequence of the very ambitious goal of giving a coherent formal basis to a large variety of optimizations in a way that fully preserves program semantics.

In our case, the definition of translation correctness is far less ambitious: preserve only the *defined* behaviors of the input program, those where no branching, memory access, or computation uses undefined values. This approach is consistent with the synchronous programming paradigm, where the absence of undefined behaviors is assumed guaranteed by program analyses performed at high abstraction level (*e.g.,* in Lustre). Under this paradigm, `undef` can be correctly compiled into *any legal value of the target SSA dialect*. In particular:

- Replacing `undef` with any defined value (*e.g.,* the constant 0) allows translation to standard SSA, but introduces initializations that do not exist in the original specification and which may hamper optimization.
- Replacing it with either `llvm.undef` or `llvm.poison` requires targeting LLVM IR and its SSA extensions, but does not introduce semantically unneeded initializations and enables LLVM-level code transformations/optimizations relying on undefinedness.

To formally state and prove translation correctness, assume that Int^* is the semantic domain of each variable, obtained by extending $\overline{Int} = Int \cup \{\perp\}$ (defined in Section 2.1) with values specific to the translation target. For instance, for a translation to LLVM IR, we shall set $Int^* = \overline{Int} \cup \{\texttt{llvm.poison}, \texttt{llvm.undef}\}$. We endow Int^* with the partial order determined by $\perp \leq x$ for all $x \neq \perp$. This partial order extends pointwise to partial valuations of the variables, and then to execution states. Then, we have:

THEOREM 3.1. *Consider a non-reactive SSA function*[9] f *and consider the result* f' *of replacing all instances of* `undef` *in* f *with values of* $Int^* \setminus \{\perp\}$ *or with other variables (without breaking dominance).*[10] *Consider an initial execution state* s_0 *and an execution trace* $s_0 \to \ldots \to s_n$ *of* f *that does not block. Then, there exists a unique trace* $s_0 = s'_0 \to \ldots \to s'_n$ *of* f' *such that* $s_i \leq s'_i$ *for all* i.

Proof sketch: Given that the trace of f does not block, differences between s_i and s'_i can only arise at the level of `undef` operations of the original program which are replaced with defined values or with `llvm.undef` or `llvm.poison`. These differences can only be

[9]That may use `undef`, but not `tick`, cyclic I/O, or synchronous modularity.
[10]f and f' will have the same variables.

propagated through assignment, as they do not reach decisions, memory access addresses, or computations. □

3.5 Formal semantics of reactive extensions

The final step of our SSA extension is the definition of the formal semantics of reactive SSA specifications formed of one or more instances of reactive SSA functions.

Given a reactive function `f`, we denote with $in^{\mathtt{f}}$, respectively $out^{\mathtt{f}}$ the ordered set of input channel variables of `f`. Given an instance i, we denote with $r(i)$ its reactive function.

The **execution state** of a reactive SSA specification is represented with triples $< i, \mathcal{I}, m >$, where m is the shared memory state, i is the currently active instance identifier, and $\mathcal{I}$ is a map associating to each instance identifier the instance state. The state of instance i is $\mathcal{I}(i) = (s, cs, si, so)$, where s is the state of the function `f` that is currently executing, cs is the call context, $si : in^{r(i)} \to \overline{Int}$ is the state of the input channels of i, and $so : out^{r(i)} \to \overline{Int}$ is the state of the output channels.

$$\frac{\mathcal{I}(i) = (s, cs, si, so) \qquad (s, cs, m) \to (s', cs', m')}{< i, \mathcal{I}, m > \Rightarrow < i, \mathcal{I}[i \leftarrow (s', cs', si, so)], m' >} \textbf{ (local)}$$

$$\frac{\mathcal{I}(i) = (Run^{\mathtt{f}}(l, s), cs, si, so) \qquad op(l) = \text{“}v = \mathtt{input}(ic)\text{“}}{< i, \mathcal{I}, m > \Rightarrow < i, \mathcal{I}[i \leftarrow (Run^{\mathtt{f}}(nxt(l), s[v \leftarrow si(ic)]), cs, si, so)], m >} \textbf{ (in)}$$

$$\frac{i = [] \quad \mathcal{I}(i) = (Run^{\mathtt{f}}(l, s), cs, si, so) \quad op(l) = \text{“}\mathtt{tick}\text{“}}{< i, \mathcal{I}, m > \xRightarrow[si']{so} < i, \mathcal{I}[i \leftarrow (Run^{\mathtt{f}}(nxt(l), s), cs, si', \bot)], m >} \textbf{ (system-tick)}$$

$$\frac{\mathcal{I}(i) = (Run^{\mathtt{f}}(l, s), cs, si, so) \qquad op(l) = \text{“}\mathtt{output}(oc : v)\text{“}}{< i, \mathcal{I}, m > \Rightarrow < i, \mathcal{I}[i \leftarrow (Run^{\mathtt{f}}(nxt(l), s), cs, si, so[oc \leftarrow s(v)])], m >} \textbf{ (out)}$$

$$\frac{\begin{array}{c}\mathcal{I}(i) = (Run^{\mathtt{f}}(l, s), cs, si, so) \qquad op(l) = \text{“}(v_1, \ldots, v_n) = \mathtt{inst\ nd}\ i'\ (w_1, \ldots, w_k)\text{“} \\ \mathcal{I}(i' :: i) = (Run^{\mathtt{g}}(l', s'), cs', si', so')\end{array}}{< i, \mathcal{I}, m > \Rightarrow < i' :: i, \mathcal{I}[(i' :: i) \leftarrow (Run^{\mathtt{g}}(l', s'), cs', \bot[in_l^{\mathtt{nd}} \leftarrow s(w_l) \mid 1 \le l \le k], \bot)], m >} \textbf{ (inst)}$$

$$\frac{\begin{array}{c}\mathcal{I}(i' :: i) = (Run^{\mathtt{g}}(l', s'), cs', si', so') \quad op(l') = \text{“}\mathtt{tick}\text{“} \\ \mathcal{I}(i) = (Run^{\mathtt{f}}(l, s), cs, si, so) \quad op(l) = \text{“}(v_1, \ldots, v_n) = \mathtt{inst\ nd}\ i'\ (w_1, \ldots, w_m)\text{“}\end{array}}{< i' :: i, \mathcal{I}, m > \Rightarrow < i, \mathcal{I}[i \leftarrow (Run^{\mathtt{f}}(l, s[v_i \leftarrow so'_i \mid 1 \le i \le n]), cs, si, so)], m >} \textbf{ (inst-tick)}$$

Fig. 11. SSA semantics extension for reactive synchronous systems. In red, modularity rules.

Under these notations, the operational semantic rules are provided in Fig. 11. The (local) rule transforms non-reactive SSA transitions of the instances (denoted with $\to$ and defined in Figures 3 and 9) into transitions (denoted with $\Rightarrow$) of the reactive system. All other rules involve reactive operations (interaction with the scheduler). Rules (in) and (out) deal with instance I/O. Rule (system-tick) is the only one that interacts with the environment by performing I/O and possibly time synchronization. The rules (inst) and (inst-tick) (in red) implement modularity. The first one gives control to an instance when reaching an `inst` operation, and the second one recovers control from it when the execution of the instance reaches a `tick` operation.

Compilation correctness issues. While defining a formal semantics for reactive SSA specifications, this paper does not provide a formal proof of the fact that their compilation to executable code is correct. One issue here is proving that the primitives of the `sync` dialect work as intended. In turn, this requires modeling the target execution platform, which may range from bare hardware to real-time operating systems.

But even assuming that the primitives are correctly implemented, we still have to prove that correct SSA code transformations preserve the semantics of reactive specifications.

This is a key issue, because the whole point of our work is to allow existing SSA code transformations and optimizations to be applied at full strength in the compilation of reactive programs.

This problem is difficult, and involves multiple difficult sub-problems. For instance, while we provide a formal semantics for reactive SSA specifications, this semantics does not cover MLIR-specific extensions to SSA. To understand the importance of these extensions, consider the `tick()` operation. As explained in Section 3.1.1, this operation has unspecified side-effects. This is why correct SSA code transformations cannot make function calls[11] commute with `tick()` operations, and thus cannot change the allocation of function calls into execution cycles.

Side effects are specified in MLIR using operation properties called *traits*. These MLIR-specific constructs must be taken into account by any formalization of the correctness of the SSA code transformations. In other terms, our formalization of the SSA semantics must be extended, if the objective is to formally state or prove the correctness of *any* MLIR-based compilation process (including ours).

However, even though formal arguments are missing, we consider that *we have provided initial evidence that existing SSA code transformations and optimizations can be applied at their full strength in the compilation of reactive specifications.*

The first argument supporting this statement is that the (complex) use cases of Section 5 run as expected after multiple, complex MLIR code transformations and optimizations. This form of validation is akin to the traditional regression testing used to validate most compilers (MLIR included), and we must significantly extend it.

The second type of evidence consists in the careful choices we explicitly made in Section 3 in the definition of the `sync` dialect operations (*e.g.*, the fact that `tick()` has unspecified side-effects). Along with the existing SSA semantics, these choices provide support for both semi-formal reasoning, and for the future formalization of the MLIR SSA extensions.

4 EMBEDDING LUSTRE IN MLIR

In the previous section, we have extended the SSA form with the features allowing the representation of synchronous reactive behaviors. Our objective was to do this in the most general way that remains fully compatible with traditional SSA semantics and algorithmics (thus allowing implementation without changes to the existing codebase).

In this section, we evaluate the ability of this extension to support the specification and compilation of *realistic* applications with both reactive and HPC aspects. The reactive aspects of such applications must be specified in a high-level synchronous dialect, not directly in our low-level SSA extension. We show that the dataflow core of the Lustre synchronous language[12] can be embedded as a new *dialect*, named `lus`, into the SSA-based MLIR compilation framework [17]. This allows the specification (in MLIR) of applications where the reactive aspects are modeled at the Lustre abstraction level, while data processing is modeled at the abstraction level of other dialects such as `affine` (affine loops), `linalg` (linear algebra), or `tf` (TensorFlow graphs).

During compilation, reactive statements of the `lus` dialect are *lowered*[13] into a mix of structured control flow (MLIR dialect `scf`) and the reactive SSA constructs introduced in Section 3 (Fig. 4) and grouped in a new dialect named `sync`. Operations of the `sync` dialect

[11] Which implicitly have side-effects in MLIR.
[12] Chosen for its practical importance, as well as for its simplicity.
[13] I.e., transformed into code at a lower abstraction level.

are later converted into low-level SSA (dialects `std` and `llvm`) and calls to external OS primitives following the compilation rules defined in Section 3.

The structured control flow, along with the data types and data processing code, are progressively lowered using the existing transformations of MLIR, which do not affect reactive semantics. Among others, these transformations allow *buffer synthesis*, i.e. the transformation of the abstract aggregate data used with `lus`-level synchronous specification (e.g., tensors) into memory objects along with allocation and deallocation operations, in a way that ensures the absence of both memory leaks and accesses to unallocated memory.

The result is a **fully functional specification and compilation framework for reactive high-performance applications**, which we evaluate in the next section on three non-trivial applications (a signal processing vocoder and two machine learning applications).

4.1 The Lustre language

The synchronous language that has reached the most widespread use is Lustre [3, 12]. For space reasons, we only consider here its pure dataflow core—the SN-Lustre dialect of [6]—into which full Lustre can be translated.

A Lustre program, like that of Fig. 2(left), is called a *node*. It specifies a dataflow graph of statements connected through dataflow variables. Each variable is either an input of the node, or it is output of exactly one statement (*single assignment property*). Lustre follows a *pure dataflow* paradigm, with no use of load/store memory (no side-effects).

The semantics of Lustre is (intuitively) best described as a mix of dataflow operational and declarative aspects.

4.1.1 Operational interpretation. As synchronous programs, Lustre nodes have a cyclic execution model. At each execution cycle, the list of statements of a node is traversed *once*, in an order compatible with the dependences determined by the variables. When traversed, a statement will read the value of its input variables, possibly perform some internal computations, and assign a value to its output variables. As the semantics of a node is not affected by the syntactic order of its statements, we can always assume (as a normal form assumption) that the statements already are in traversal order, as in Fig. 2(left).

Dependences are defined as follows: Variable `y` being produced by statement p and used in statement c determines a dependency $p \rightarrow c$ in all cases except one: when c is a statement of the form "`x = k fby y`". In this case, $c \rightarrow p$, as a form of anti-dependency ensuring that the value is read before being overwritten. This special handling of `fby` allows it to read the value assigned to its input variable *in the previous execution cycle*. This value is then assigned to the output of `fby`, allowing its use in the current cycle. In Lustre, this is the only mechanism allowing the specification of a *state* passed between execution cycles.

The value assigned to a variable in a cycle can be the special value $\perp$ introduced in Section 2.1 to represent undefinedness/absence. Making a variable absent inside an execution cycle is done using the sub-sampling statement `when`. Traversing statement "`x = y when c`" while the Boolean variable `c` has value true results in `x` being assigned the value of `y`. If `c` is false or $\perp$, `x` is assigned value $\perp$. Absence is the dataflow mechanism allowing the specification of conditional execution: A function call with $\perp$ input variables will not perform computations, but instead assign all outputs to $\perp$. This mechanism, specific to the dataflow programming paradigm, is used in lines 13 and 14 of Fig. 2(left) to specify that function `actuate` is executed only in cycles where variable `ck` is true. The counterpart of `when` is statement "`x = merge c y z`". Upon traversal, if `c` is true (resp. false), `x` takes the value of `y` (resp. `z`). Otherwise, `x` takes the value $\perp$.

```
1  node s(i:int)returns(s:int)
2  var ck:bool;i1,s1:int;
3  let
4
5    ck = (i>0) ;
6    i1 = i when ck ;
7    s1 = 0 fby s ;
8    s  = s1 + i1 ;
9
10 tel
```

```
1  node sn(i:int)returns(s:int)
2  var ck:bool;i1,s1,si,so:int;
3  let
4    si = 0 fby so;
5    ck = (i>0);
6    i1 = i when ck;
7    s1 = si when ck;
8    s  = s1 + i1;
9    so = merge ck s (si whenot ck);
10 tel
```

Fig. 12. Time modularity (left) and its normalization

4.1.2 Declarative aspects (clock constraints). The purely operational interpretation defined above matches the standard Lustre semantics [3, 4, 6] on correct Lustre programs where all `fby` statements are executed at every cycle.

However, Lustre allows a form of logical time modularity during specification, by allowing sub-sets of statements (including `fby` statements) to be executed under specific Boolean conditions. This is the case in the node of Fig. 12(left), which incrementally computes the sum of its *positive* input values. The statements in lines 7 and 8, which implement the computation of the sum, are executed only in cycles where `ck` is true. However, under the dependency rules defined above, no operational mechanism constrains the execution of the `fby` statement.

Instead, each Lustre statement defines a *clock constraint*—a predicate relating the present/absent status of its input and output variables (and possibly their value) inside an execution cycle. In our example, two such clock constraints apply:

- Inputs and outputs of a function call or `fby` are either all $\perp$, or all have a value different from $\perp$.
- The inputs of a `when` are either both present, or both absent. Output is present *iff* the condition is true.

Determining the execution condition of each statement consists in solving the system of constraints associated to all statements. This process,[14] called *clock inference* and performed in formal settings called *clock calculi*, has been the subject of extensive prior research [2, 4, 12].

4.2 The `lus` dialect

We have embedded the Lustre language constructs into MLIR. This allows the MLIR-level representation of both general and normalized Lustre specifications. The `lus` representation of the Lustre node in Fig. 12(right) is provided in Fig. 13(left), with the `lus` operations highlighted in blue. The MLIR definition of nodes is derived from that of SSA functions by replacing keyword `func` with `node` and requiring the compiler to check that a single basic block is present, terminated with operation `lus.yield` (which identifies the output variables).

While MLIR is SSA-based and its transformations mostly require the respect of dominance, it also allows specification under the weaker single assignment property, thus allowing the representation of cyclic dataflow graphs like the one in Fig. 13(left).

At the level of `lus`, we have implemented the Lustre clock inference algorithm. Along with data type correctness, single assignment, and causal correctness, the success of clock

[14] Which will fail for incorrect programs.

```
node @s2(%i:i32)->(int){

  %c0=constant 0:i32

  %si=fby %c0,%so:i32

  %ck=cmpi "sgt",%i,%c0:i32
  %i1=when %ck,%i:i32
  %s1=when %ck,%si:i32

  %s=addi %s1,%i1:i32

  %s2=when not %ck,%si:i32
  %so=merge %ck,%s,%s2:i32

  lus.yield %s:i32
}
```

```
1  func @s2(in(i32),out(i32))->(){
2  ^start(%ic:in(i32),%sc:out(i32)):
3    %c0=constant 0:i32
4    br ^step(%c0:i32)
5  ^step(%si:i32):
6    %i = input(%ic):i32
7    %ck=cmpi "sgt",%i,%c0:i32
8    %i1=when %ck,%i:i32
9    %s1=when %ck,%si:i32
10
11   %s=addi %s1,%i1:i32
12   output(%sc:%s):i32
13   %s2=when not %ck,%si:i32
14   %so=merge %ck,%s,%s2:i32
15
16   tick()
17   br ^step(%so:i32)
18 }
```

```
1  func @s2(in(i32),out(i32))->(){
2  ^start(%ic:in(i32),%sc:out(i32)):
3    %c0=constant 0:i32
4    br ^step(%c0:i32)
5  ^step(%si:i32):
6    %i = input(%ic):i32
7    %ck=cmpi "sgt",%i,%c0:i32
8    cond_br %ck,^aux(%i:i32,%si:i32),
9      ^out(%si:i32)
10 ^aux(%i1:i32,%s1:i32):
11   %s=addi %s1,%i1:i32
12   output(%sc:%s):i32
13   br ^out(%s:i32)
14
15 ^out(%so:i32):
16   tick()
17   br ^step(%so:i32)
18 }
```

Fig. 13. MLIR lowering of `lus`(left) to `sync`(right). In the middle, the output of the first lowering phase.

inference guarantees the correctness of a Lustre program. As data type correctness and single assignment are automatically checked by the existing MLIR infrastructure, only causal correctness remains to be checked.

4.2.1 Normalization. Once clock inference performed, we can transform the original program so that the operational interpretation correctly simulates it. This transformation, exemplified in Fig. 12, consists in ensuring that all `fby` statements are executed at every cycle (through the introduction of `when` and `merge` statements, in red in Fig. 12). We have implemented this normalization step in MLIR.

4.2.2 Lowering of `lus` to `sync`. After normalization, moving from the dataflow style of `lus` to the control-flow style of the `sync` dialect of Section 3 (Fig. 4) is done in two steps, as exemplified in Fig. 13:

- Moving from dataflow node-based modularity to control-flow function-based modularity.
- Encoding of conditional execution and absence.

The first step starts by collectively replacing all `fby` operations with the continuation passing encoding of state specific to MLIR SSA. In Fig. 13(middle) this is done using the basic branching mechanism of SSA, but the lowering phase we implemented in MLIR produces structured control flow operations (`for`, of dialect `scf`). Once the state reencoding performed, determining the causal correctness of the program amounts to an SSA dominance check, performed automatically by MLIR.

Then, the I/O variables of the `lus` node are transformed into I/O channels and explicit `input` and `output` operations, and the node interface is transformed into a function interface. This concludes the first lowering step.

Once the first step is done, the semantic rules of Figures 3, 9, and 11 can already be applied, and the need to re-encode signal absence becomes evident. Indeed, in the definition of the Lustre semantics, we have made the implicit assumption that operation call semantics is changed to produce $\perp$ output upon receiving $\perp$ input (and not block execution). This is acceptable from a semantic point of view (the Merge operation of TensorFlow already uses a similar mechanism [22]). However, from an implementation standpoint this would

require wrapping every data type to account for the extra $\perp$ value, and wrapping function to ensure that undefined inputs result in absence of execution. To avoid this (for efficiency), the dataflow control transmitted by the $\perp$ values is materialized under the form of branching code that determines, based on the value of clock variables (e.g., `%ck` in Fig. 13) whether operations are executed or not. In Fig. 13(right) this imperative control is implemented using core SSA branching statements, but our MLIR implementation uses structured control flow operations.

The clock analysis of the Lustre program ensures that the resulting SSA specification cannot block under blocking semantics due to operations receiving undefined values[15]; proving this result goes beyond the scope of this paper, but the reader may refer to [3, 4] for formalization and proofs in the classical Lustre setting.

This completes the definition of our compilation method. Note that, in the implementation of our compiler, stock MLIR algorithms are used for all data type specification and implementation, causality analysis, and memory allocation. We have only had to implement analysis and lowering steps specific to the synchronous model of computation—clock analysis, normalization, and the lowering of normalized `lus` to `sync`.

5 USE CASES

In this section, we use three complex applications to illustrate the expressiveness of our MLIR embedding of Lustre, and to evaluate its effectiveness. The applications have been selected to cover various profiles involving both performance-intensive and reactive aspects:

Vocoder is a real-time sound processing application (a voice pitch tuning vocoder). This application, which relies on classical signal processing algorithms like the Fast Fourier Transform (FFT) is representative for traditional performance-intensive signal processing applications. It also features complex reactive control: a sliding window over the inputs, a keyboard-driven human-computer interface allowing muting the sound output and changing its volume, and a stateful signal processing algorithm.

TimeSeries is a recurrent neural network (RNN) based on LSTM [14]. To faithfully implement the LSTM stateful behavior in a streaming application, this application features multi-rate execution. This application is representative for convolutional neural networks (CNNs) with complex control.

Resnet50 is an instance of the classical ResidualNet [13], representative for large feed-forward deep neural networks (DNNs).

All three applications have non-trivial data processing pipelines and are naturally expressed using the reactive primitives of the `lus` dialect and the data-processing features of existing MLIR dialects.

To evaluate the *expressiveness* of our MLIR embedding of Lustre we show that, for all three use cases, MLIR extended with the `lus` dialect allows the natural and concise expression of all application aspects (data processing and reactive ones). This means that the modeling approach we propose is more adapted to high-performance embedded applications than either MLIR alone (which lacks reactive specification constructs) and various dialects of Lustre, which lack advanced tensorial data specification and manipulation routines, as well as the efficient compilation apparatus.

Part of the *effectiveness* evaluation has been done in the previous section, which showed that the few Lustre-specific compilation algorithms (clock analysis, normalization, lowering of `lus` to `sync`) can be embedded into MLIR, and that stock MLIR algorithms can be

[15]They can still block due to an operation being undefined for the provided values, e.g., division by zero.

used for the remaining compilation steps: data type analysis and implementation, causality analysis, memory allocation....

The only important remaining question is whether the Lustre-specific compilation algorithms can be combined in optimized compilation flows with the advanced optimization algorithms already implemented in MLIR, which are ultimately responsible for the performance of MLIR-generated code [5]. To prove this, we have considered a sub-set of the advanced optimization methods of [5] and integrated them as steps of our `lus` compiler pipeline. Then:

- Using the ML examples (TimeSeries and Resnet50) we show that our compiler already matches (and sometimes outperforms) existing state-of-the-art compilation flows for the reactive language Lustre.
- For the vocoder example, we show that its implementation is fast enough to allow real-time execution on low-cost commercial hardware.

The full code of our compiler and of the three use cases is provided at https://github.com/dpotop/mlir-lus-public. For portability, the implementation of modularity is `longjmp`-based.

5.1 The pitch tuning vocoder

The pitch tuning vocoder cyclically samples the sound input and the other control inputs, updates the control state, performs (if required by the current control status) one step of the pitch tuning algorithm, and drives the sound output.

The pitch processing pipeline,[16] specified using the abstract aggregate types of MLIR (tensors) and operations of the `linalg`, `scf`, and `std` dialects, is encapsulated in the `@pitch_algo` node.

Fig. 14 shows the top-level node `@pitch` of the application. This node concentrates most of the reactive control features: the sliding window, multiple feedback loops, as well as the control driven by the user interface, which allows silencing the output (and shutting down unneeded pipeline steps) and altering the pitch correction. The `@pitch` node features all the operations of the `lus` dialect:

- `lus.node` and `lus.yield` to specify the node interface and its output.
- `lus.when` and `lus.merge` to enforce conditional execution.
- `lus.instance` to instantiate nodes `@pitch_algo` and `@kbd_ctrl`.

Our compilation flow automatically performs buffer allocation and synthesizes low-level control ensuring the correctness and safety of the implementation. Various MLIR-provided optimizations can be applied in the process, as discussed in Section 5.3.

5.2 Neural network use cases

We use the `lus` dialect to represent the reactive control of two convolutional neural networks: an instance of the `tf.keras.applications.ResNet50` implementation of the Residual Network [13] and a time series forecasting application[17] based on LSTM [14].

The more interesting of the two, from a reactive specification standpoint, is the time series application. Indeed, while ResNet is a feedforward network, the LSTM at the core

[16] Involving Hann filtering to reduce spectral leakage, move to frequency domain using the Fast Fourier transform (FFT), move to polar coordinates to separate magnitude from phase, the additive phase synthesis algorithm that performs the actual pitch change, move back to Cartesian coordinates and then to the time domain using the inverse FFT, and a final step of Hann filtering and additive accumulation.

[17] Full application provided (in Keras) at https://machinelearningmastery.com/tensorflow-tutorial-deep-learning-with-tf-keras/, section 3.3

```
// At each cycle, the @pitch node receives from the environment the current keypress
// (if any) and 512 half-precision integers representing 256 stereo samples.
// It produces an equal amount of processed samples.
lus.node @pitch(%kbd:i8,%sndbufin:tensor<512xi16>)->(tensor<512xi16>) {
  // The sliding window over input sound samples (initialized with 0)
  %c0 = call @bzero_i16_256() : ()->tensor<256xi16>
  %circ3 = call @stereo2mono(%sndbufin):(tensor<512xi16>)->(tensor<256xi16>)
  %circ2 = lus.fby %c0 %circ3 : tensor<256xi16>
  %circ1 = lus.fby %c0 %circ2 : tensor<256xi16>
  %circ0 = lus.fby %c0 %circ1 : tensor<256xi16>
  %window = call @concat_samples(%circ0,%circ1,%circ2,%circ3)
    : (tensor<256xi16>,tensor<256xi16>,tensor<256xi16>,tensor<256xi16>)
    -> tensor<1024xf32>
  // Keyboard controller, which sets the control state based on keyboard input
  %sndon,%volume,%semitones = lus.instance @kbd_ctrl(%kbd) : (i8)->(i1,f32,f32)
  // Pitch processing pipeline, executed only when %sndon is true
  %window_cond = lus.when %sndon %window : tensor<1024xf32>
  %semitones_cond = lus.when %sndon %semitones : f32
  %pitch_output = lus.instance @pitch_algo(%semitones_cond, %window_cond)
   :(f32,tensor<1024xf32>)->(tensor<1024xf32>)
  // Sound volume control, only executed when %sndon is true
  %vol_cond = lus.when %sndon %volume : f32
  %output = call @f32_tensor_float_product(%vol_cond,%pitch_output)
   :(f32,tensor<1024xf32>)->(tensor<1024xf32>)
  // When %sndon is false, we still have to output zero (muted sound)
  %muted = call @bzero_f32_1024(): () -> tensor<1024xf32>
  %output_merged = lus.merge %sndon %output %muted: tensor<1024xf32>
  // Extract the last 256 samples and convert them to the correct output format
  %out = call @extract_samples(%output_merged) : (tensor<1024xf32>)->(tensor<256xi16>)
  %sndbufout = call @mono2stereo(%out):(tensor<256xi16>)->(tensor<512xi16>)
  lus.yield(%sndbufout:tensor<512xi16> )
}
```

Fig. 14. Pitch control application—top-level node. In blue, `lus` dialect operations.

of the time series application is a recurrent layer. Furthermore, the behavior specified by the use case is a multi-rate one: input is cut in sections of 5 cycles each, with one output produced at the end of each interval. Subsampling is performed on the output of the `@lstm` node instance. Thus, computations driven by the output of the LSTM are executed at a pace 5 times slower than the internal computations of the LSTM (grouped in node `@lstm`). Such multi-rate execution mechanisms are also common in embedded control applications specified in Lustre, and can be used to support multi-period execution [10].

5.3 Comparison with Lustre

For all three applications, we have also encoded the same behavior in Lustre, using the open-source state-of-the-art Heptagon compiler.[18] In doing this, we took advantage of the Heptagon-specific language extensions, which allow for limited array processing using map/fold iterators, in order to improve performance.

As our use cases only feature fixed-sized data, encoding in Lustre/Heptagon posed no problem. Of course, our Lustre embedding into MLIR is more expressive in this respect, as

[18] http://heptagon.gforge.inria.fr/

it can take advantage of the full MLIR type system, which allows for arrays and tensors of dynamic sizes.

But the most interesting part of the comparison concerns the performance figures, summarized in Table 1. To obtain them, the Heptagon-generated code has been compiled with `gcc` at increasing optimization levels and executed on a Intel Core i5-7600 processor rated at 3.50GHz. The MLIR code has been compiled using two different flows: a default flow without very little MLIR-level optimization, where only common subexpression elimination is performed in addition to the mandatory lowering and normalization passes, and a high-level flow that includes loop nest optimizations representative of compilers for high-performance computing. These loop nest optimizations including loop tiling, packing (copying for cache spatial locality and conflicts) and loop unrolling; they have been implemented in MLIR within the course of a couple of weeks and do not mean to compete with the most agressive, automatically tuned optimizers for tensor computations [5]. For each use case, every 6 configurations were executed 10 times and the smallest figure has been considered in each configuration to reduce the effect of OS interference.[19]

Table 1. Average execution time per cycle for the neural network use cases

Specification&compilation method	ResNet50	TimeSeries
MLIR/lus	11.748s	250 μs
MLIR/lus optimized	4.873s	130 μs
Heptagon+gcc-O0	151.847s	846 μs
Heptagon+gcc-O1	52.730s	212 μs
Heptagon+gcc-O2	22.592s	148 μs
Heptagon+gcc-O3	5.165s	40 μs

It is no surprise that, in the absence of optimizations, Heptagon-generated code is inefficient. However, on the ResNet50 example which is representative of large ML applications, the still preliminary and lightweight MLIR loop nest optimizations yield better performance than the state-of-the-art Heptagon+gcc-O3 flow, which leverages the agressive optimizations of the (currently) most performant open-source compiler (`gcc`).

Given that we only scratched the optimization potential of MLIR [5], these results clearly show that the marriage we propose between MLIR and Lustre has the potential for high runtime performance.

Overall, this evaluation confirms the expressiveness and effectiveness claims, about leveraging existing SSA-based optimizations, the ease of implementing optimizations specific to dataflow synchronous compilers, and the ability to apply loop nest optimizations such as loop tiling to a dataflow synchronous program. This draws a very promising path for the development of specification formalisms, programming languages and compilers for applications where both performance and predictability are important.

6 RELATED WORK

We advance the state-of-the-art in SSA by providing the syntactic constructs, structural rules, and semantics extensions allowing the representation of synchronous reactive behavior. This overall objective is fully original.

However, particular aspects of our extension have been covered in previous work, in particular predicated execution [7, 18] and the use of semantic absence/undefinedness [9, 15]. In previous SSA work, predicated execution is *explicit*: the predicates are represented at

[19]Our experiments do not involve random elements, so that we assumed that OS-related interferences are responsible for most of the variance.

all points where they exert control. We allow an *implicit* predication specification, akin to dataflow semantics, where the absence of inputs determines absence of execution. This requires a clear definition of correctness, which is absent in [7].

Our treatment of semantic absence/undefinedness is also different from that proposed for LLVM [15]. Our objective is not to ensure that non-deterministic code (involving undefined values) preserves its set of traces unchanged under various optimizations. Instead, it is to guarantee that well-defined values are preserved by various optimizations, even in the presence of undefined behaviors affecting other variables. We attain this goal through the abstraction theorem of Section 3.4.

We also extend previous work on the compilation of synchronous languages [2, 4, 19]. Existing synchronous language compilers profoundly restructure the code in order to match an execution model (which, as explained in Section 3.1.2, is in most cases sequential function calls, even though multi-threaded implementations have also been proposed [16]). We fully avoid this by ensuring that every reactive SSA specification can be executed *as is*, and be subject to any correct SSA transformation. Synchronous-specific code transformations are still possible, but not mandatory.

More important maybe, instead of advancing the compilation of synchronous languages as a separate research field, we show that it can be seen as an extension of classical compilation, allowing the reuse of fundamental techniques for type checking, causality analysis, buffer allocation *etc.*

7 CONCLUSION

We presented an embedding of the Lustre synchronous reactive language into SSA, extending the MLIR framework to allow synchronous programming and code generation. We illustrated it by capturing the data processing, computational and reactive control aspects of signal processing and ML applications. Our MLIR extension remains fully compatible with SSA-based analysis and transformations while also capturing the synchronous composition of concurrent state machines, logical and physical time synchronization, and compilation passes specific to synchronous languages, such as clock calculus, causality analysis, initialization analysis, bounded-memory and clock-directed code generation.

While there had long been connections between the semantics of functional languages, SSA and dataflow synchronous languages, this paper describes the concrete extension of SSA capturing the necessary elements for reactive control system. Furthermore, retaining all SSA-based compilation algorithms and reusing existing code unaware of the the specific synchronous concurrency semantics allows a tighter integration of reactive system modeling frameworks with the computationally intensive and general-purpose capabilities of MLIR- and LLVM-based frameworks for HPC and ML.

Our work paves the way for unified design and implementation flows where the specification of reactive control, state machines, and real-time ressource orchestration become first class citizens on par with parallelization and optimization in general-purpose frameworks. One key objective here is to lift our unification between reactive control and data parallelism from the level of the MLIR intermediate format (not meant for direct programming) to that of high-level specification languages such as Lustre, Keras or Jax. Since synchronous concurrency is highly popular in safety-critical environments, one important future direction is to fully formalize the core MLIR components our extension relies upon, and to prove its correctness. A related direction consists in exploring syntax and refinements for state-machines and control automata expressed in the Esterel synchronous language as well as hardware design languages such as Bluespec and Chisel. It is also important to further

investigate which SSA-based algorithms can benefit the compilation flow of reactive control systems and how domain-specific methods for synchronous languages can enable greater automation and guarantees in the composition and memory management of concurrent systems in general.

REFERENCES

[1] Arinc 653: Avionics application software standard interface. part 1 - required services. revision 3, 2010.

[2] Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., LeGuernic, P., and de Simone, R. The synchronous languages 12 years later. *Proceedings of the IEEE 91*, 1 (Jan 2003).

[3] Bergerand, J., Caspi, P., Pilaud, D., Halbwachs, N., and Pilaud, E. Outline of a real time data flow language. In *Proceedings RTSS* (San Diego, CA, USA, December 1985).

[4] Biernacki, D., J.-L. Cola c., Hamon, G., and Pouzet, M. Clock-directed modular code generation for synchronous data-flow languages. In *Proceedings LCTES* (2008).

[5] Bondhugula, U. High performance code generation in mlir: An early case study with gemm, 2020.

[6] Bourke, T., Brun, L., Dagand, P.-E., Leroy, X., Pouzet, M., and Rieg, L. A formally verified compiler for lustre. In *Proceedings PLDI* (2017).

[7] Carter, L., Simon, B., Calder, B., Carter, L., and Ferrante, J. Predicated static single assignment. In *Proceedings PACT* (1999).

[8] Cytron, R., Ferrante, J., Rosen., B., Wegman, M., and Zadeck, F. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst. 13*, 4 (1991).

[9] Demange, D., and Y. Fernández de Retana, D. P. Semantic reasoning about the sea of nodes. In *Proceedings CC* (Vienna, Austria, 2018).

[10] Didier, K., Potop-Butucaru, D., Iooss, G., Cohen, A., Souyris, J., Baufreton, P., and Graillat, A. Correct-by-construction parallelization of hard real-time avionics applications on off-the-shelf predictable hardware. *ACM Trans. Archit. Code Optim. 16*, 3 (2019), 24:1–24:27.

[11] Gérard, L., Guatto, A., Pasteur, C., and Pouzet, M. A modular memory optimization for synchronous data-flow languages: application to arrays in a Lustre compiler. In *Proceedings LCTES* (2012).

[12] Halbwachs, N. A synchronous language at work: the story of Lustre. In *Proceedings Memocode* (Verona, Italy, 2005).

[13] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016).

[14] Hochreiter, S., and Schmidhuber, J. Long short-term memory. *Neural Computation 9*, 8 (1997), 1735–1780.

[15] Lee, J., Kim, Y., Song, Y., Hur, C.-K., Das, S., Majnemer, D., Regehr, J., and Lopes, N. Taming undefined behavior in llvm. In *Proceedings PLDI* (2017).

[16] Li, X., and von Hanxleden, R. Multithreaded reactive programming-the kiel esterel processor. *IEEE Transactions on Computers 61*, 3 (2012).

[17] Multi-level intermediate representation compiler framework (MLIR). https://mlir.llvm.org/, Retrieved on 11/17/2020.

[18] Ottenstein, K., Ballance, R., and MacCabe, A. The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. *SIGPLAN Not. 25*, 6 (June 1990), 257–271.

[19] Potop-Butucaru, D., Edwards, S., and Berry, G. *Compiling Esterel*. Springer, 2007.

[20] Rosen, B., Wegman, M., and Zadeck, F. Global value numbers and redundant computations. In *Proceedings POPL* (Jan 1988).

[21] Static single assignment book (in progress). http://ssabook.gforge.inria.fr/latest/book.pdf, Retrieved on 11/17/2020.

[22] The TensorFlow Authors. Implementation of control flow in tensorflow (online pdf). Nov 2017.