In-Place Update in a Dataflow Synchronous Language: A Retiming-Enabled Language Experiment

Ulysse Beaugnon
ENS
ulysse.beaugnon@ens.fr

Albert Cohen
ENS, INRIA
albert.cohen@inria.fr

Marc Pouzet
ENS, INRIA
marc.pouzet@ens.fr

ABSTRACT

Dataflow synchronous languages such as LUSTRE have a purely functional semantics. This incurs a high overhead when dealing with arrays, as they have to be copied at each update. We propose to tackle this problem at the source, by constraining programs so that every functional array definition can be optimized into an in-place update. Our solution handles aliasing between function arguments. It also allows more programs with in-place updates to be accepted thanks to a new retiming framework, effectively rescheduling computations across time steps. Our proposed language and compilation method enforces zero-copy purely functional arrays while preserving expressiveness and programmer control through explicit copies.

CCS Concepts

•Software and its engineering → Functional languages; Data flow languages;

Keywords

Synchronous Language, Lustre, Array, In-Place Update, Functional Semantics

1. INTRODUCTION

Dataflow synchronous languages such as SIGNAL [11], LUSTRE [9] and SCADE [17] are used to design and implement certified control software. They are based on a discrete notion of time that allows them to describe interactions between the program and its environment and to guarantee the reaction time to external stimuli [4]. Their simple and purely functional semantics makes the correctness of dataflow synchronous programs easier to check than with traditional languages. However, this purely functional semantics is a source of compilation difficulties in presence of large data structures such as arrays. Indeed, purely functional data structures are immutable and any modification must occur to a copy of the structure to avoid modifying the original. Thus, to update a single cell of an array, a costly copy of the whole array must be performed.

This paper makes two complementary contributions.

- We add constraints on accepted programs to *allow every functional array definition to be implemented as an in-place update* in memory. The programmer may still insert explicit copies when needed to relax scheduling constraints, but copying and memory usage overheads will never be a surprise to the embedded system designer. While the idea is well-known in the context of conventional functional languages [10], the originality of our approach is to use scheduling constraints to relax the compilation of array definitions as in-place updates, and to only reject programs where ordering constraints are incompatible. Unlike array optimizations in traditional functional languages, we take advantage of the lack of explicit ordering of array definitions in a dataflow language.
- We consider two application scenarios: one may either eliminate all copies by allowing arrays to propagate across time steps, or one may copy arrays at the end of each time step to ensure that memory-based dependences (i.e., dependences induced by memory reuse) do not propagate to a different step. The first scenario is more permissive as it accepts programs with less copies, yet it requires modeling dependencies across time steps. We achieve this through an extension of the modular static scheduling algorithm of Pouzet and Raymond [15]. We call this new algorithm *modular retiming.* The second scenario is simpler and conservative w.r.t. current synchronous compilers. It allows static memory allocation of arrays, as live ranges are bounded by the end of each step.

Our contributions remain of an exploratory nature. In particular, a thorough investigation of the language's expressivess remains to be conducted, comparing with array copy optimizations for dataflow synchronous languages [7] and with the expressiveness of (unsafe) imperative languages with arrays. Nevertheless, while detailing the formalism and algorithms, we will illustrate our solution and validate its effectiveness on simple examples, capturing classical array computations patterns.

2. SYNCHRONOUS LANGUAGES

Dataflow synchronous languages have been designed to program safety-critical control systems. They are based on

SCOPES '16, May 23-25, 2016, Sankt Goar, Germany
© 2016 ACM. ISBN 978-1-4503-4320-6/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2906363.2906379

```
(* a node with one input and two outputs*)
node foo(a: int) = (b: int, c: int)
  { (* body of foo *) ...  }

(* call to foo *)
b: int, c:int = foo(123)

(* ∀t ∈ ℕ. a(t) = b(t) + c(t-1) *)
a: int = b + pre c

(* Computes a(t) = t *)
a: int = 0 -> pre a + 1

(* Assigns b or c to a depending on b *)
a: int = if b then c else d

(* Declare an array filled with 0 *)
A: int[8] = 0^8

(* Read A *)
a3: int = A[3]

(* B is equal to A, with A[3] set to 5 *)
B: int[8] = {A with [3] = 5}

(* C is a copy of A *)
C: int[8] = copy A
```

Figure 1: The toy language

a discrete notion of time that allows them to describe the interaction of the system with its environment, and to bound the reaction time of systems to external stimuli [4].

A synchronous program is composed of a set of functions called *nodes*. Each node contains mutually recursive equations that define sequences of values. The sequences of values correspond to the evolution of variables at each time step. Equations may only reference values produced at the current or the previous time step. The compiler is responsible for ordering equations according to data dependencies and for producing a *step* function that evaluates the equations of a node at a given time step to compute the next value of each sequence.

For the purpose of this paper, we present a simplified synchronous dataflow language based on LUSTRE. This language does not feature constructs such as clocks or automata. The ability to deal with conditional control flow in our toy language indicates that such important constructs should combine naturally with our proposal, but the actual integration is left for future work.

Variables are either of scalar type `int` or `bool` or of array type `int[size]` or `bool[size]`. The size of arrays must be known statically. The value of variables at the previous times step can be accessed using the `pre` operator. `->` is used to initialize variables at the first time step. Figure 1 shows a few constructs of the toy language we use in this paper.

An interesting feature is the possibility to write feedback loops: an input of a node may depend on an output produced by the same node. The code in Figure 2 shows an example of such a feedback loop.

Before any analysis is performed, equations are decomposed into elementary operations by introducing temporary variables (Figure 3).

3. ZERO-COPY ARRAYS

```
(* a is used as input to foo but
 * depends on its output b *)
b: int, c: int = foo(a)
a: int = b + 2
```

Figure 2: Feedback loop

```
(* complex equation *)
a: int = b + (c * d)

(* decomposed equations *)
tmp: int = c * d
a: int = b + tmp
```

Figure 3: Code turned into elementary equations.

In a traditional synchronous dataflow language, the equation `B: int[8] = {A with [i] = x}` would let `A` intact and create a copy of `A` to store `B`. Copy operations can be costly compared to in-place updates, especially for large arrays. However, if the compiler can ensure that `A` is not accessed after `B` is created, `A` can be used instead of its copy. We thus add extra scheduling constraints on accepted programs to ensure that no array is accessed after it has been written to. In practice, it boils down to add dependencies from reads to writes to the same array and to ensure no array is written to twice. For example, in Figure 4, `d` is ordered before `C` so that the definition of `C` is an in-place update.

```
C: int[8] = {A with [7] = 2} (* Write A *)
d: int = A[3]                (* Read A *)
```

Figure 4: Reads are ordered before writes

This technique to enforce in-place updates is well-known in conventional functional languages [10]. The originality of our approach is to exploit the degrees of freedom offered by dataflow languages, where equations are ordered by the compiler and not by the programmer.

If a program cannot be scheduled to fulfill the constraits imposed by in-place updates, explicit copies have to be inserted. This way, the programmer knows exactly where copies happen and he can work to remove them. An exemple of program that needs an explicit copy can be found in Figure 5.

We also want to avoid copies when assigning an array to another, for example when writing `A: int[8] = B`. In such a case, we say that `A` and `B` *alias* with each other. This means `A` and `B` should be considered as the same array when adding scheduling constraints or ensuring no array is written twice. Aliasing also occurs when choosing between two arrays with an `if` condition and when passing arrays as arguments to functions. In other words, aliasing refers to the functional-style aliasing of array names, and not to a points-to relation related to some memory representation of these arrays. Of course, aliased arrays always share the same memory location, but this can also be the case for two arrays with one defined as an in-place update of the other. For example, array `A` and `B` in Figure 5 above share the same memory location but do not alias. A full description of how we compute aliasing is available in Section 4.2. The aliasing on

```
  (* One of the writes to A must copy it *)
  B: int[8] = {A with [i] = x}
  C: int[8] = {copy A with [j] = y}
```

Figure 5: Program that needs an explicit copy

function arguments require special care as different calling contexts might create different aliasing schemes between the arguments. We show how to deal with this problem in Sections 4.3 and 5.

There may also be aliasing between different steps. Consider, for example, the program in Figure 3. If `pre` does not copy arrays but creates aliasing instead, the array stored in `A` at step t may stay indefinitely alive in `B` and `C`. Its memory thus cannot be reused for `A` in subsequent steps. New memory has to be dynamically allocated for `A` at each step.

```
A: int[8] = x^8
B: int[8] = if c then A else C
C: int[8] = 0^8 -> pre B
y: int = B[i]
```

Figure 6: Aliasing accross time steps.

We consider two situations. In the first one, we allow aliasing across time steps. This reduces the number of necessary copies, but requires a dynamic allocation scheme for arrays. In the second one, the delay operator `pre` is considered to copy its array argument. It avoids aliasing across time steps and is suitable for systems that cannot afford dynamic allocation.

A side-effect of the aliasing across time steps is that some dependencies can also cross time steps boundaries and may require the evaluation of some equations to be delayed or advanced to another time step. This is similar to retiming in synchronous circuits [12]. In Section 5 we present a framework to handle retiming in a modular way in dataflow synchronous languages.

4. SCHEDULING CONSTRAINTS

This section details the construction of scheduling constraints to ensure no array is accessed after it has been written to. Section 4.1 shows how dependencies are represented, Section 4.2 how the aliasing information is computed and Section 4.3 how to compile a node without the aliasing information from its calling context.

4.1 The Dependency Relation

Consider a node N in the dataflow program, with a set of variables V, a set of inputs $I \subseteq V$ and a set of outputs $O \subseteq V$. We want to compile N into a step function that processes a single time step for N. This step function should be the same for all time steps. We thus look for dependency information independent of the current time step. For this purpose, we define the dependency relation $\prec$ such that $\forall w \in \mathbb{Z}, a, b \in V$:

$$\begin{aligned} b \overset{w}{\prec} a \quad &\iff \quad \forall t \geq 0 : a(t+w) \text{ is scheduled after } b(t) \\ &\iff \quad \exists t \geq 0 : a(t+w) \text{ depends on } b(t) \end{aligned}$$

As dependencies are transitive, $\prec$ verifies:

$$a \overset{w}{\prec} b \wedge b \overset{w'}{\prec} c \implies a \overset{w+w'}{\prec} c \qquad (1)$$

To compile N as a step function that computes the value of each variable at the next time step, we must also enforce $a(t)$ to be computed before $a(t+1)$ $\forall a \in V$ with the constraint:

$$\forall a \in V : a \overset{1}{\prec} a \qquad (2)$$

Definition 1. *$\prec$ is the smallest relation in $V \times \mathbb{Z} \times V$ verifying (1) and (2) such that*

- *data dependencies are respected: if* `a = op(..., b, ...)` *then $b \overset{0}{\prec} a$ and if* `a = pre b` *then $b \overset{1}{\prec} a$.*
- *if $a, b \in V$ are tow arrays such that:*

  $$\exists t \geq 0.\ a(t) \text{ alias with } b(t+w)$$

  and c, d two equations such that c reads a and d reads b, then $c \overset{w}{\prec} d$.

The next definition characterizes nodes that should be accepted by the compiler. Proposition 1 shows that this definition matches exactly the nodes that can be compiled into a step function.

Definition 2. *N is* causally correct *if $\forall a, b \in V, w \in \mathbb{Z}$:*

- *Their is no dependency cycle: $a \overset{w}{\prec} a \implies w > 0$.*
- *Retimings are bounded: $\forall a, b \in V$, $\exists w \in \mathbb{Z} : \neg \left(a \overset{w}{\prec} b \right)$.*

To simplify notations, we also define the *depends-or-equals* relation $\preccurlyeq$ as:

$$a \overset{w}{\preccurlyeq} b \iff a \overset{w}{\prec} b \vee (a = b \wedge w \geq 0)$$

Remark 1. *When N is causally correct, $\prec$ can be recovered form $\preccurlyeq$.*

To compute $\prec$, we represent it on a weighted directed graph G whose vertices are the variables of N and whose edges represent the dependencies directly induced by data dependencies and read-write pairs to aliasing arrays. For each $v \in V$, we also have an edge $a \xrightarrow{1} a$ to accommodate (2). Their is a path from a to b with weight w in G if and only if $a \overset{w}{\prec} b$.

Remark 2. *Let $a, b \in V$ and $w \in \mathbb{Z}$. Then:*

$$a \overset{w}{\prec} b \iff \forall w' \geq w : a \overset{w'}{\prec} b$$

Remark 2 tells us that we can compute $\prec$ only by looking at shortest paths in G, which can be done in polynomial time.

Function calls are handled using a summary of the dependency graph. The summary is also a graph, but only contains the input and output variables. For $i \in I, o \in O$ such that the shortest distance from i to o in G is w, there is an edge $i \xrightarrow{w} o$ in the summary. The existence of the shortest distance is enforced by the causality of $\prec$. The summary is inlined into the dependency graph of each calling context and has exactly the same behavior as the full dependency graph of the node with regard to the computation of $\prec$ in the calling context.

Equation	Edges
`a = b`	$b \xrightarrow{0} a$
`a = if x then b else c`	$b \xrightarrow{0} a \wedge c \xrightarrow{0} a$
`a = pre b`	$b \xrightarrow{1} a$
`a = b -> c`	$b \xrightarrow{0} a \wedge c \xrightarrow{0} a$

Table 1: Ancestor graph edges

4.2 The Aliasing Relation

We are interested in dependencies of the form:

$$\exists t \geq 0 : \ a(t) \text{ depends on } b(t-w)$$

We thus consider an aliasing relation $a \overset{n}{\sim} b$ with a and b equations and $n \in \mathbb{Z}$ such that:

$$\exists t : \ a(t) \text{ aliases with } b(t+n) \implies a \overset{n}{\sim} b$$

The aliasing relation is symmetric ($a \overset{w}{\sim} b \iff b \overset{-w}{\sim} a$) and can only be an over-approximation. It is used both to order reads and writes to aliasing arrays and to ensure that no array is written twice. The approximation presented here is elementary as aliasing analysis is not the main subject of this paper. It is based on the notion of ancestors: two variables alias with each other if their value may come from the same variable. Ancestors are computed from a weighted directed graph whose nodes are the variables and whose edges represents the origins of variables. Their is a path $a \xrightarrow{n}^* b$ if $a(t)$ is an ancestor of $b(t+n)$. The edges of the ancestor graph are defined by Table 1. Once ancestors are known, the aliasing relation is computed as:

$$a \overset{m-n}{\sim} b \iff \exists s \in V. \ s \xrightarrow{n}^* a \wedge s \xrightarrow{m}^* b \qquad (3)$$

As we are only looking for dependencies with a minimal weight (Remark 2), we are also looking for aliasing with minimal weight. Thus, when looking for n (resp. m) such that $s \xrightarrow{n}^* a$ (resp. $s \xrightarrow{m}^* b$), we only need to look for a path of minimal (resp. maximal) weight. This allows to compute the necessary aliasing information in polynomial time.

Similarly to what we did for the dependency relation, we use a summary of the ancestor graph to propagate aliasing information accross node calls. The summary contains the input and output variables plus a vertex $v_{o_1 o_2}$ for all pairs of outputs $o_1, o_2 \in O$. The edges are defined as follows:

- For $i \in I, o \in O$ such than m_0 and m_1 are respectively the minimal and maximal distances from i to o in the ancestor graph, two edges $i \xrightarrow{m_0} o$ and $i \xrightarrow{m_1} o$ are added to the summary.
- For $o_1, o_2 \in O$ such that $o_1 \overset{m}{\sim} o_2$ with m minimal, two edges $v_{o_1 o_2} \xrightarrow{-m} o_1$ and $v_{o_1 o_2} \xrightarrow{0} o_2$ are added to the summary.

The case where their is no minimal or maximal distance between two variables in the aliasing graph is handled by extending the weigh on the aliasing relation to $\mathbb{Z} \cup \{+\infty, -\infty\}$. $a \overset{-\infty}{\sim} b$ (resp. $+\infty$) means the weights on the aliasing between a and b are not lower (resp. upper) bounded.

If we want to avoid aliasing across time steps, we make all delay operators such as `pre` copy their argument. That is, if `x` is an array, `pre x` is treated as `pre(copy(x))`. The third line of Table 1 is removed, in which case the aliasing relation can only have a weight of zero (and thus weight can be ignored). From that it follows that we can no longer create dependencies $a \overset{w}{\prec} b$ with $w < 0$ so retiming is no longer needed.

4.3 Aliasing from the Calling Context

The aliasing in a node may depend on its calling context. Consider, for example, the code in Figure 7. The aliasing between `A` and `B` depends on the aliasing between the arguments passed to `foo`. This is a problem as we need this information to order `d` (that reads `A`) and `C` (that writes to `B`).

```
node foo(A: int[8], B: int[8]) = (e: int) {
  C: int[8] = {B with [0] = 42}
  d: int = A[3]
  e: int = d + C[3]
}
```

Figure 7: Aliasing depends on the calling context.

We solve this problem by transfering the responsability to order such read-write pairs to the calling context. We expose reads from arrays as dummy inputs and writes to arrays as dummy outputs. When a node is instantiated, the calling context expresses the dependency from a read to a write by adding a dependency from the dummy output (corresponding to the read) to the dummy input (corresponding to the write). The problem is thus reduced to the already existing problem of compiling feedback loop.

Figure 8 shows the node `foo` from the code above. The read to `A` and the write to `B` are exposed as an input and an output. The dependency from `d` to `C` (in red) can be added by the calling context by adding a feedback loop from `Read A` to `Write B` (in blue).

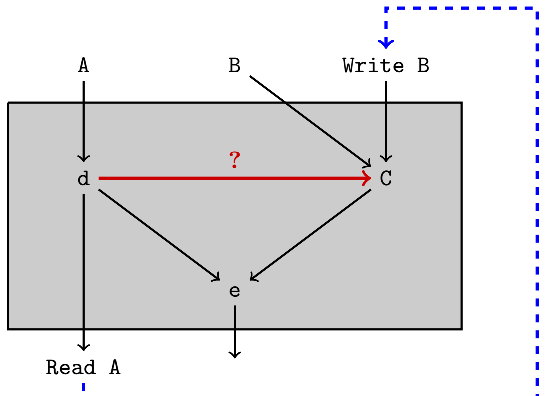

Figure 8: Node `foo` dependency graph.

Actually, not every array access needs to be exposed to the calling context. We present here a few rules to reduce the number of exposed accesses, ensuring it remains linear in the original number of inputs and outputs.

- Only accesses to arrays that alias with an input or an output need to be exposed. As there is at most one write per array, this limits the number of writes exposed to the number of inputs and outputs.

- Arrays can only be defined once. If a variable only aliases with arrays that are being written to, it cannot be a source of read-write dependency anymore. Thus reads to such variables do not need to be exposed.
- Read to arrays can be grouped per input or output they alias with. Indeed, if a write is performed on an array which aliases with an input or an output, dependencies are added from all the reads to arrays that alias with this input or output. Such a read can be treated as a single source of dependencies. As an array may alias with multiple inputs or outputs, a read may be exposed multiple time. This rule ensures that the number of reads exposed is linear in the original number of inputs or outputs.

With this section on aliasing, the compilation issues raised by zero-copy arrays are reduced to scheduling problems. In the case where aliasing across time steps is forbidden, retiming is not needed and usual existing scheduling techniques used in synchronous compilers can be used. The next section discusses the more general situation where aliasing is considering accross several time steps and, thus, retiming is needed.

5. MODULAR RETIMING

To limit the size of the generated code and to be able to compile each node only once and independently of the calling context, the compiler should not inline nodes in their calling context. In dataflow synchronous languages, avoiding inlining is not trivial. Indeed, the presence of feedback loops in the calling context might add dependencies from outputs to inputs in the called node. Two different calling contexts might impose incompatible scheduling constraints on a node. It is thus not possible to compile a node into a single step function that can be called indifferently from any calling context, unless a unit delay is added on every feedback loop.

Pouzet and Raymond have studied this problem and proposed a solution called *gray-boxing* [15] following that of Raymond [16]. This approach is halfway between a full inlining and the compilation of the node into a single step function. For each node, a few sub-nodes are created. Each sub-node is a group of equations that can always be executed atomically, irrespectively of any dependency imposed by the calling context. Sub-nodes can be compiled to sequential code like conventional functions. The calling context only reorders sub-nodes to satisfy the dependency it adds to called nodes. For example, on Figure 9 where black edges represent data dependencies, three sub-nodes can be created.

In order to minimize the size of the generated code, the number of sub-node should be minimal. However this problem has been shown to be NP-complete [13, 15]. Pouzet and Raymond provide a heuristic to find a good partitioning in polynomial time; otherwise, an algorithm that finds an optimal partitioning using a SAT-solver can be used.

In the case where we forbid aliasing across time steps the approach of Pouzet and Raymond can be used to allow modular scheduling. However their method does not handle retiming and thus cannot be used if we have aliasing across steps. This section presents an extension of their solution to

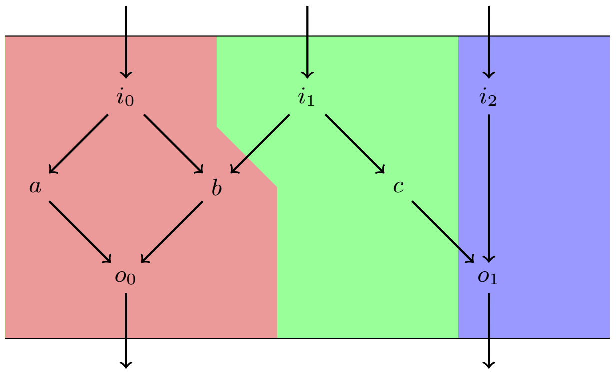

Figure 9: Partitioning of a node into sub-nodes.

handle retiming. Our proposal follows the same structure as theirs. First, we formalize the non-modular scheduling problem in presence of retiming; then we show how the gray-boxing technique can be applied to it. Next, we present a heuristic to find a good partitioning in sub-nodes and last an algorithm that finds an optimal partitioning quantifier-free Presburger arithmetic solver.

5.1 Static Schedule

In this section, we define the notion of *static schedule* which is a total order between equations together with a retiming function. A single step function is generated for each node, obtained by statically scheduling equations. This solution imposes that every feedback loop crosses an explicit delay.

DEFINITION 3. *A* static schedule $(r, <)$ *is composed of:*

- *A retiming function* $r : Eq \to \mathbb{Z}$ *to choose which variables are computed in a given step. The value of* $a(t)$ *will be computed at the step* $t + r(a)$*.*
- *A total order* $<$ *on equations to give the evaluation order inside the step function.*

A schedule is correct *if it respects dependencies:*

$$a \overset{w}{\prec} b \implies r(a) < r(b) + w \lor (r(a) = r(b) + w \land a < b)$$

PROPOSITION 1. *A node has a static schedule if and only if it is causally correct (See 2).*

Algorithm 1 gives a static schedule. It is used to schedule equations within each sub-nodes and to schedule the top node.

PROPOSITION 2. *If run on a causally correct node, Algorithm 1 terminates and produces a correct static schedule.*

5.2 A Gray-Boxing Formalization

In this section we formalize the idea of gray-boxing in presence of retiming. If the dependencies across time steps are ignored (i.e. if all dependencies have a null weight) this formalization corresponds exactly to the one given by Pouzet and Raymond for the case without retiming.

Algorithm 1: Non-modular scheduling

```
Data: The variables $V$ and the dependency relation $\prec$
Result: A static schedule $(r, <)$
/* Compute retiming                                  */
foreach $v \in V$ do
    $r(v) \longleftarrow 0$
while $\exists a, b \in V.\ a \overset{w}{\prec} b \wedge r(b) - r(a) + w < 0$ do
    $r(b) \longleftarrow r(a) - w$
/* Compute $<$, seen as a subset of $V \times V$     */
$< \longleftarrow \emptyset$
foreach $a, b \in V$ do
    if $a \overset{n}{\prec} b \wedge r(b) - r(a) + n = 0$ then
        $< \longleftarrow < \cup (a, b)$
Complete the partial order $<$ using topological sort
return $(r, <)$
```

5.2.1 Weighted Preorder

We first define the notions of weighted preorder and weighted equivalence. These two relations are the extension with weight of the preorder and equivalence relations used in the case without retiming.

Definition 4. *A* weighted preorder $\overset{n}{\precsim}$ *over a set S is a relation $\subseteq S \times \mathbb{Z} \times S$ such that:*

- *It is reflexive with any positive weight: $\forall n \geq 0 : a \overset{n}{\precsim} a$.*
- *It is transitive: $a \overset{n}{\precsim} b \wedge b \overset{m}{\precsim} c \implies a \overset{n+m}{\precsim} c$.*

Moreover, a weighted preorder is said to be valid iff weights have a lower bound for each pair of elements:

$$\forall a, b \in S\, \exists n_0 \in \mathbb{Z} : a \overset{n}{\precsim} b \implies n \geq n_0$$

Definition 5. *A* weighted equivalence *relation $\overset{n}{\simeq}$ over a set S is a ternary relation $\subseteq S \times \mathbb{Z} \times S$ such that:*

- *It is reflexive: $a \overset{0}{\simeq} a$.*
- *It is transitive: $a \overset{n}{\simeq} b \wedge b \overset{m}{\simeq} c \implies a \overset{n+m}{\simeq} c$.*
- *Weights are unique: $a \overset{n}{\simeq} b \wedge a \overset{m}{\simeq} b \implies n = m$.*
- *Weights are negated when the operands are swapped: $a \overset{n}{\simeq} b \iff b \overset{-n}{\simeq} a$.*

Similarly to regular equivalence relations, we can define equivalence classes. Two nodes x and y are in the same equivalence class iff $\exists n : x \overset{n}{\simeq} y$. To build a weighted equivalence, a weighted preorder can be used.

Proposition 3. *A valid weighted preorder $\precsim$ induces a weighted equivalence $\simeq$ defined by:*

$$a \overset{n}{\simeq} b \iff a \overset{n}{\precsim} b \wedge b \overset{-n}{\precsim} a \tag{4}$$

5.2.2 Gray-Boxing Induced by a Weighted Preorder

We now show how a valid weighted preorder $\precsim$ that respect dependencies (i.e., $a \overset{n}{\prec} b \implies a \overset{n}{\precsim} b$) can be used to define a partitioning into sub-nodes.

Let $\simeq$ be the weighted equivalence relations induced by $\precsim$ and $X_0, \ldots, X_{k-1}$ its equivalence classes. Each equivalence class X_i represents a different sub-node.

Let $(r_i, <_i)$ be the static schedule associated with each X_i. The retiming r_i can only be defined up to an additive constant. Indeed, the sub-node itself will be retimed once instantiated in the calling context. We thus choose an arbitrary equation $a \in X_i$ and define r_i relatively to $r(a)$. $\forall b \in X_i$, there is a unique $n \in \mathbb{N}$ such that $b \overset{n}{\simeq} a$ thus we can define $r_i(b)$ by $r_i(b) = r_i(a) + n$. Once the retiming is given, $<_i$ is computed as in Algorithm 1.

Remark 3. *r_i does not depend on the choice of a.*

Now that we have the schedule of each sub-node, we can define $\prec_X$ to represent the dependencies between each pair of sub-nodes. $\prec_X$ will be used by the calling context to order sub-nodes. $\preccurlyeq_X$ is actually easier to define. This is not a problem as $\prec_X$ can be recovered from it (Remark 1). $\preccurlyeq_X$ is defined by:

$$\forall i, j \in [\![0,\, k-1]\!] : X_i \overset{n}{\preccurlyeq}_X X_j$$
$$\iff \exists a_i \in X_i,\, a_j \in X_j : a_i \overset{n+r_i(a_i)-r_j(a_j)}{\precsim} a_j$$

This definition ensures dependencies among equations are respected since $\precsim$ contains the dependency relation $\prec$.

Remark 4. *To compute $\prec_X$, it is enough to choose one representative from each equivalence class and to compute the dependencies from them. Indeed, as $\precsim$ is transitive:*

$$\exists a_i \in X_i,\, a_j \in X_j : a_i \overset{n+r_i(a_i)-r_j(a_j)}{\precsim} a_j$$
$$\iff \forall a_i \in X_i,\, a_j \in X_j : a_i \overset{n+r_i(a_i)-r_j(a_j)}{\precsim} a_j$$

Reciprocally, the original valid weighted preorder can be recovered from the gray boxing. Indeed, let $X_0, \ldots, X_{k-1}$ be a partitioning of equations, r_i the retiming inside each sub-node and $\preccurlyeq_X$ the depends-or-equals relation among sub-nodes. Consider $\precsim$ defined by:

$$a \overset{n}{\precsim} b \iff a \in X_i \wedge b \in X_j \wedge X_i \overset{m}{\preccurlyeq}_X X_j \wedge n \geq m + r(b) - r(a)$$

Then, $\precsim$ is a weighted preorder inducing the gray-boxing. It means that by looking at valid weighted preorder we are looking at every possible gray-boxing.

5.2.3 Static Partitioning

To ensure the gray-boxing of a node does not restrict its possible calling contexts, we introduce the notion of *static partitioning*. As we will show below in Proposition 4, a static partitioning is exactly a preorder that respects dependencies and does not add constraints on possible calling contexts. The definition and the proposition given here are transpositions of the ones given by Pouzet and Raymond on preorders and equivalence relation to weighted preorder and weighted equivalence.

Definition 6. *A* static partitioning *is a valid weighted preorder* $\overset{n}{\precsim}$ *such that:*

- *It contains all the dependency constraints:*

$$x \overset{n}{\preccurlyeq} y \implies x \overset{n}{\precsim} y \tag{5}$$

- *It strictly maps dependencies on input/output pairs:*

$$\forall i \in I, o \in O:\ i \overset{n}{\preccurlyeq} o \iff i \overset{n}{\precsim} o \tag{6}$$

Moreover, a static partitioning is optimal iff its induced equivalence relation $\overset{n}{\simeq}$ *has a minimal number of classes.*

Proposition 4 shows the static partitionings are exactly the preorders that do not reject any valid context.

Proposition 4. *Let N be a node and $\precsim$ be a valid weighted preorder respecting the dependencies of N. Then $\precsim$ is a static partitioning if and only if for every calling context C of N, the node identical to C but where the call to N has been replaced by the gray-boxing induced by $\precsim$ is also causally correct.*

Remark 5. *The depends-or-equals relation $\preccurlyeq$ is a static partitioning. It corresponds to the gray-boxing where each equation is a separate box, or in other word the full inlining of equations in the calling context.*

5.2.4 Compatibility

To give hints on how to group equations for a gray-boxing, we define the compatibility relation. This relation is an extension with a weight of the compatibility relation defined by Pouzet and Raymond.

Definition 7. *The* compatibility *relation* $\overset{n\in\mathbb{Z}}{\chi}$ *of a node with inputs I, outputs O and dependency relation $\preccurlyeq$ is defined by:*

$$x \overset{a}{\chi} y \iff \forall i \in I, o \in O:\ \left(i \overset{b}{\preccurlyeq} x \wedge y \overset{c}{\preccurlyeq} o \implies i \overset{a+b+c}{\preccurlyeq} o\right)$$
$$\wedge \left(i \overset{b}{\preccurlyeq} y \wedge x \overset{c}{\preccurlyeq} o \implies i \overset{-a+b+c}{\preccurlyeq} o\right)$$

As shown below in Proposition 5, the compatibility is a necessary condition to group equations together. However, it is not sufficient to prove a gray-boxing is correct. Static partitionings should be used instead for this purpose.

Proposition 5. *If $\overset{n}{\precsim}$ is a static partitioning and $\overset{n}{\simeq}$ its associated weighted equivalence, then:*

$$x \overset{n}{\simeq} y \implies x \overset{n}{\chi} y$$

5.3 Heuristic

To find a good partitioning in polynomial time, we may use a heuristic. This heuristic starts with the depends-or-equals relation (which is a static partitioning according to Remark 5) and iteratively adds constraints to it to create a static partitioning with a small number of sub-nodes. The added constraints are based on the fact that it is always possible to group together equations depending on the same inputs or that have the same set of outputs depending on it. This heuristic is a weighted extension of the heuristic proposed by Pouzet and Raymond.

For the heuristic to work, each equation should depend at least on one input and have one output that depends on it. Equations that do not match those criteria are detected and handled separately: for equations that have no output, they can be removed. For equations which do not depend on any input, they are gathered into an additional sub-node of the gray-boxing.

We first define the *input* and *output saturations* that given a static partitioning add constraints to group together equations that depend on the same inputs or that have the same set of outputs depending on it.

Definition 8. *Given a static partitioning $\precsim$, its input function $\mathcal{I}^{\precsim}_{n}$ (resp. output function $\mathcal{O}^{\precsim}_{n}$) and its* input saturation $\precsim_{\mathcal{I}}$ *(resp.* output saturation $\precsim_{\mathcal{O}}$*) are defined as follows:*

$$\begin{aligned}
\mathcal{I}^{\precsim}_{m}(x) &= \left\{(i,n) \in I \times \mathbb{Z} \;\middle|\; i \overset{n+m}{\precsim} x\right\}\\
x \overset{m}{\precsim}_{\mathcal{I}} y &\iff \mathcal{I}^{\precsim}_{0}(x) \subseteq \mathcal{I}^{\precsim}_{m}(y)\\
\mathcal{O}^{\precsim}_{m}(x) &= \left\{(o,n) \in O \times \mathbb{Z} \;\middle|\; x \overset{n+m}{\precsim} o\right\}\\
x \overset{m}{\precsim}_{\mathcal{O}} y &\iff \mathcal{O}^{\precsim}_{m}(x) \supseteq \mathcal{O}^{\precsim}_{0}(y)
\end{aligned}$$

Proposition 6 shows the input and the output saturations produce valid static partitionings that contains all the constraints of the initial static partitioning. Moreover, the produced static partitionings are respectively optimal on outputs and on inputs, as their induced equivalence relations meet the compatibility relation on these sets of variables. Any pair of inputs (resp. outputs) that could be grouped together according to the compatibility relation are put in the same sub-node by the output saturation (resp. input saturation).

Proposition 6. *$\precsim_{\mathcal{I}}$ and $\precsim_{\mathcal{O}}$ are valid static partitionings that contains $\precsim$:*

$$\forall x, y \in V, n \in \mathbb{Z}:\ x \overset{n}{\precsim} y \implies x \overset{n}{\precsim}_{\mathcal{I}} y \wedge x \overset{n}{\precsim}_{\mathcal{O}} y$$

Moreover, $\precsim_{\mathcal{I}}$ meets the compatibility relation χ on outputs and $\precsim_{\mathcal{O}}$ on inputs:

$$\left(\forall o_1, o_2 \in O:\ o_1 \overset{n}{\chi} o_2 \iff o_1 \overset{n}{\simeq}_{\mathcal{I}} o_2\right)$$
$$\wedge \left(\forall i_1, i_2 \in I:\ i_1 \overset{n}{\chi} i_2 \iff i_1 \overset{n}{\simeq}_{\mathcal{I}} i_2\right)$$

As $i \overset{n}{\precsim} i \implies i \overset{n+1}{\precsim} x$, $\mathcal{I}^{\precsim}_{m}(x)$ can be efficiently represented by only storing the minimal n such that $i \overset{n}{\precsim} x$ for each $i \in I$. The same goes for $\mathcal{O}^{\precsim}_{m}(x)$.

Definition 9. *Let $\precsim$ be a static partitioning. Its* input/output saturation $\precsim_{\mathcal{I}_{\mathcal{O}}}$ *is defined as the input saturation of its output saturation.*

As the input and the output saturations preserve their original preorder, the input/output saturation combines the benefits of both. It is a static partitioning that contains $\precsim$ and is optimal on both inputs pairs and outputs pairs. Proposition 7 shows it is also optimal on input-output pairs.

PROPOSITION 7. *The input/output-saturation meets the compatibility on input-output pairs:*

$$\forall i \in I, o \in O : \; i \overset{n}{\chi} o \iff \overset{n}{\simeq}_{\mathcal{I}_\mathcal{O}} o$$

In the end, we propose a *modular retiming* heuristic that is optimal on $I \cup O$. Its behavior on its internal equations remains to be validated on a panel of real-world applications. However, the results we have obtain on small examples and the fact that it is a generalization of an existing heuristic gives us good hopes on its effectiveness. Moreover, a sub-optimal solution increases the size of the generated code but does not alter the correctness or the genericity of a node.

Our heuristic runs in time $O\left(|V|^3\right)$. This complexity is due to the fact that we compute the shortest path between all pair of variables on the dependency graph to build the input and the output saturations.

5.4 An Exact Algorithm

For completeness, one may also look for a minimal static partitioning. The definition of a static partitioning with at most k equivalence classes is encoded as a formula of the quantifier-free Presburger arithmetic. Finding an assignment of the variables that satisfy such a formula is NP-complete, but we cannot do better as our problem is already known to be NP-hard, even without taking retiming into account [15, 13].

A valid preorder $\precsim$ over a set of variables V is encoded by two sets of variables:

- c_{ab}, $a, b \in V$: a boolean that indicates whether b depends on a:

$$c_{ab} \iff \exists n \in \mathbb{Z} : \; a \overset{n}{\precsim} b$$

- w_{ab}, $a, b \in V$: an integer which has a meaning only when $c_{ab} = \mathit{true}$, it indicates the minimal n such that $a \overset{n}{\precsim} b$. Such a n exists since $\precsim$ is assumed valid.

The reflexivity and the transitivity of the preorder are respectively encoded in (7) and (8):

$$\bigwedge_{a \in V} \quad c_{aa} \wedge w_{aa} \leq 0 \tag{7}$$

$$\bigwedge_{a,\, b,\, c \in V} \quad (c_{ab} \wedge c_{bc}) \implies (c_{ac} \wedge w_{ac} \leq w_{ab} + w_{bc}) \tag{8}$$

The restrictions for the preorder to be a static partitioning are encoded in:

$$\bigwedge_{\substack{a \overset{n}{\prec} b \\ n \text{ minimal}}} c_{ab} \wedge w_{ab} \leq n \quad \bigwedge_{\substack{i \overset{n}{\prec} o \\ n \text{ minimal}}} w_{io} = n \quad \bigwedge_{\nexists n :\, i \overset{n}{\prec} o} \neg c_{io} \tag{9}$$

To count the number of classes, a variable x_a to indicate the class of each $a \in V$ is used. (10) ensures those variables are coherent with the preorder and (11) that there are at most k classes.

$$\bigwedge_{a,\, b \in V} \quad x_a = x_b \implies (c_{ab} \wedge c_{ba} \wedge w_{ab} = -w_{ba}) \tag{10}$$

$$\bigwedge_{a \in V} \quad 0 \leq x_a < k \tag{11}$$

An assignment to the variables that satisfies the conjunction of (7), (8), (9), (10) and (11) yields a static partitioning with at most k equivalence classes. To ensure k is minimal, we try to find a solution with $k = 1$, and then increase k until a solution is found. To find an assignment of variables for a given k, any off-the-shelf SMT solver with support for the quantifier-free Presburger arithmetic can be used.

6. EXAMPLES

We first present an implementation of a Finite Impulse Response (FIR) filter with zero-copy arrays (Figure 10). The input signal `a` is stored in a circular buffer X and the response b is computed using coefficients $f_0, \ldots f_4$. The program has a single node, and two arrays `X` and `Y`, defined inductively. If aliasing across time steps is forbidden, it is possible to generate the `C` code shown in Figure 10. The write to `X` is compiled into an in-place update. However, `pre Y` requires a copy operation. On the other hand, if we allow aliasing across time steps, the copy operation is replaced with a plain `pre_Y = Y`. Now `X`(t) alias with `Y`$(t-1)$, which induces a dependency from `b`(t) that reads `Y`(t) to `Y`$(t+1)$ that defines `X`$(t+1)$.

```
node filter(a: int) = (b: int) {
  X: int[4] = 0^n -> pre Y
  Y: int[4] = {X with [i] = a}
  i: int = 0 -> pre i+1 mod 4
  b = Y[i]*f0          + Y[i-1 mod 4]*f1
    + Y[i-2 mod 4]*f2 + Y[i-3 mod 4]*f3
}

// Generated C code
X = pre_Y;
i = (i+1) % 4;
Y = X;
Y[i] = a;
b = Y[i]*f0        + Y[(i+1)%4]*f2
  + Y[(i+2)%4]*f2 + Y[(i+3)%4]*f3;
memcopy(pre_Y, Y, 4*sizeof(int));
```

Figure 10: Dataflow implementation and compilation of a FIR filter

This first example did not feature the need for retiming or for gray-boxing. Let us consider a second example, in Figure 11, in which both are needed. In the node `assign`, both arrays `A` and `B` are inputs, so the write to `A` should be exposed as an input and the read from `B` as an output as represented on Figure 12. Gray-boxing yields two subnodes: S_0 (in red) and S_1 (in blue) with the dependency $S_0 \overset{0}{\prec} S_1$.

```
node assign(A: int[8], B: int[8]) = (C: int[8]) {
  b3: int = B[3]
  C: int[8] = {A with [i] = b3}
}

(* Calling context *)
E: int[8] = assign(D, pre D)
```

Figure 11: Node that needs greyboxing.

If we allow aliasing across time steps, the calling context (Figure 11) induces the alias relation $A \overset{-1}{\sim} B$ in `assign`. A dependency `Read B` $\overset{-1}{\prec}$ `Write A` which results in $S_0 \overset{-1}{\prec} S_1$ is thus added, which in turn forces S_0 to be delayed by at least one step with regard to S_0. Compilation to in-place updates may now proceed without further retiming.

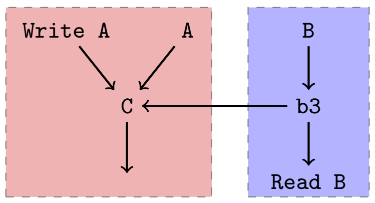

Figure 12: Dependencies of the assign node.

7. RELATED WORK

The problem of eliminating array copies is not limited to dataflow synchronous languages. It occurs in every language that uses purely functional data structures, and it has thus been widely studied [10]. The proposed approaches fall into three categories.

The first category uses persistent data structures [6]. Such data structures are engineered to keep the history of modifications that occurred to the data structure instead of requiring a whole copy. Data structures for persistent arrays have been proposed [5]. However, the cost of maintaining the data structure and of finding the right version at each access incurs an overhead.

The second category relies on static analyses to remove as many copies and fresh array allocations as possible. The compiler looks for the last use of each variable and implements in-place updates when arrays are not live anymore. Methods relying on abstract interpretation [14], heuristics [18], or Hindley-Milner type inference [2] have been proposed to compute the live range of arrays. In conjunction with static approaches, dynamic mechanisms such as reference counting are used for cases that cannot be decided statically. The work reported in [1] presents an instance of this idea applied to the dataflow synchronous language LabView. They also use scheduling constraints to allow in-place updates, but only as an optimization, when the compiler can prove it will not add dependency cycles. Theses approaches allow some operations to be performed in-place without changing the semantics of the language. However, they do not ensure that *all* updates are performed in-place, and they are rather brittle as a small change to the program may introduce a copy operation without warning (e.g., making an array update conditional).

The last category makes in-place update the default, only accepting programs where data structures can indeed be updated in-place. Such solutions are usually based on linear logic [8] to limit variables to a single use, thus ensuring they are never used after being written to. In practice, this limitation is too strong and several propositions relaxed the linearity constraint [19, 3]. In [7], the authors apply this idea to a Lustre-like language; they propose an annotation mechanism to specify which arrays should share the same memory location and use a type system based on linear logic to ensure the soundness of annotations. It handles inter-procedural copy-avoidance and guarantees no hidden copy will occur. The main differences with our work are that it requires the programmer to manually annotate every array, that it is less generic as different calling contexts might generate different aliasing configurations between arguments, which in turn would require different annotations; and finally, it is also overly restrictive as the memory location assigned to each array variables cannot be data dependent. Experimental comparisons remain to be conducted however.

8. CONCLUSION AND FUTURE WORK

We presented a new method to efficiently handle functional arrays in a synchronous dataflow language. The method automatically introduces scheduling constraints on dataflow equations, enforcing that no hidden array copy may occur in the generated code. It handles aliasing between node arguments and only requires minimal programmer intervention while preserving a purely functional semantics.

Scheduling constraints are not only used to reject programs where copies cannot be removed. They also provide the necessary information for the compiler to reorder equations. Indeed, unlike eager functional languages where the schedule is fixed in the sequential program semantics, dataflow synchronous compilers are in charge of ordering equations and scheduling elementary computations. This is a notable improvement over existing solutions where a schedule is picked by the programmer irrespectively of the array update constraints. It also improves upon the genericity of dataflow nodes, as the schedule may be dependent on aliasing between arguments, and the compiler may automatically adapt to different contexts while a programmer would have to write multiple version of the same node.

Regarding future work, our method is currently limited by the power of the alias analysis. As a result, some programs are rejected while a valid schedule would normally exist, forcing the programmer to introduce spurious copies. Our current implementation uses dynamic allocation, which can be a problem for critical systems. This requirement can be lifted by copying all arrays at the end of each time step. While this induces a high cost, it still is a good improvement over the state of the art as no array needs to be copied within a time step. In addition, even with dynamic allocation, array objects can be managed and recycled at compilation time, scanning array variables at the end of each time step to check which arrays remain accessible. Also, the compatibility of our solution with features usually found in synchronous dataflow languages, such as clock types guarding the execution of specific instructions, should be further investigated.

9. REFERENCES

[1] S. Abu-Mahmeed, C. McCosh, Z. Budimlić, K. Kennedy, K. Ravindran, K. Hogan, P. Austin, S. Rogers, and J. Kornerup. Scheduling tasks to maximize usage of aggregate variables in place. In O. de Moor and M. Schwartzbach, editors, *Compiler Construction*, volume 5501 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2009.

[2] H. G. Baker. Unify and conquer. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, Nice, France, 1990.

[3] E. Barendsen and S. Smetsers. Uniqueness type inference. In *Programming Languages: Implementations, Logics and Programs*, pages 189–206. Springer, 1995.

[4] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous

languages 12 years later. *Proceedings of the IEEE*, 91(1), Jan 2003.

[5] P. F. Dietz. Fully persistent arrays (extended array). In *Proceedings of the Workshop on Algorithms and Data Structures*, WADS '89, 1989.

[6] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. In *Eighteenth Annual ACM Symposium on Theory of Computing*, STOC '86, Berkeley, California, USA, 1986.

[7] L. Gérard, A. Guatto, C. Pasteur, and M. Pouzet. A modular memory optimization for synchronous data-flow languages: Application to arrays in a lustre compiler. In *International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, LCTES'12, Beijing, China, 2012.

[8] J.-Y. Girard. Theoretical computer science. In *Linear logic*, volume 50, pages 1–102, 1987.

[9] N. Halbwachs. A synchronous language at work: the story of lustre. In *International Conference on Formal Methods and Models for Co-Design*, MEMCODE'05, Verona, Italy, 2005.

[10] P. Hudak and A. Bloss. The aggregate update problem in functional programming systems. In *Symposium on Principles of Programming Languages*, POPL '85. ACM, 1985.

[11] P. LeGuernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with signal. *Proceedings of the IEEE*, 79(9), Sep 1991.

[12] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1), 1991.

[13] R. Lublinerman, C. Szegedy, and S. Tripakis. Modular code generation from synchronous block diagrams: Modularity vs. code size. In *Symposium on Principles of Programming Languages*, POPL '09, Savannah, GA, USA, 2009.

[14] M. Odersky. How to make destructive updates less destructive. In *Symposium on Principles of Programming Languages*, POPL '91, Orlando, Florida, USA, 1991. ACM.

[15] M. Pouzet and P. Raymond. Modular static scheduling of synchronous data-flow networks – an efficient symbolic representation. In *International Conference on Embedded Software*, EMSOFT'09, Grenoble, France, 2009.

[16] P. Raymond. Compilation séparée de programmes Lustre. Technical report, Projet SPECTRE, IMAG, July 1988.

[17] Scade website. http://www.esterel-technologies.com/products/scade-suite/.

[18] P. Schnorf, M. Ganapathi, and J. L. Hennessy. Compile-time copy elimination. *Software: Practice and Experience*, 23(11), 1993.

[19] P. Wadler. Linear types can change the world. In *IFIP TC*, volume 2, pages 347–359. Citeseer, 1990.