Efficient arrays for dataflow synchronous languages

Ulysse Beaugnon
Supervised by Albert Cohen and Marc Pouzet
Team PARKAS (ENS)

August 25, 2014

General context

Dataflow synchronous languages such as SIGNAL, LUSTRE, or their industrial counterpart SCADE are widely used to design embedded systems. They are built around 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. Their well-defined and purely functional semantics makes the correctness of dataflow synchronous programs easier to check.

The model-checking and the control flow optimization of dataflow synchronous languages have been extensively studied, but only a few publications mention optimizations for arrays.

Problem studied

Since dataflow synchronous languages are purely functional, arrays are constant and cannot be modified in-place. They have to be copied at each update. This is an increasing source of performance problems as the demand for more complex computation involving large arrays in embedded systems rises.

Some solutions have been proposed to allow in-place updates while keeping a functional semantics, but they either rely on the programmer to do the work through an annotation mechanism or they just *do their best* with no guarantee every copy will be removed. The absence of guarantee is a problem as it makes it hard to understand and solve performance issues. The annotation solution lacks in genericity as different calling contexts might need different annotations.

Proposed contributions

First we present a novel solution to ensure every array is updated in-place while keeping a functional semantics by using destructive updates. This approach has been studied in the context of conventional functional languages. It is here reformulated in terms of scheduling constraints, taking advantage of the equational, implicitly scheduled principle of dataflow languages.

With this solution, the evaluation order of the instructions inside a function depends on its calling context. A method called *grey-boxing* allows a function to be split into several sub-functions such that the content of each sub-function can always be executed in the same order. This allows a function to be compiled separately from its calling context, but finding an optimal grey-boxing is NP hard. We show the notion of grey-boxing can be formalized and generalized to handle a broader class of scheduling problems. Then we provide an exponential algorithm to find an optimal grey-boxing and a heuristic to find a good grey-boxing.

Arguments supporting their validity

Our treatment of arrays gives to the programmer a total control of where in-place updates happens without the need for additional annotations. It handles inter-procedural optimizations in a modular way that does not reduce the genericity of nodes. Moreover, it allows more copies to be removed than previous methods. Thus, we believe it is a good improvement over the existing literature.

The modular scheduling algorithm is formalized and the proofs of the correctness of both the optimal scheduling algorithm and the scheduling heuristic are given. The quality of the heuristic has not been tested yet since the implementation of our algorithms is still in progress, but several clues hint it should be satisfactory. First, it is proved to be optimal on inputs and outputs, and second, it is built over a heuristic for a simpler problem that has been shown to be efficient.

Summary and future work

First, we give some background on dataflow synchronous languages (Section 1) and we review the existing literature on optimisation for arrays in functional and dataflow synchronous languages (Section 2). Then, we propose a method to ensure every definition of an array element is done in-place (Section 3). The idea is not new and has already been extensively studied. Our main contribution here is to use the flexibility of the implicit scheduling to add dependency constraints instead of using a typing system. Finally, we show how to compile a function in a context-independent way (Sections 4 and 5). This is not trivial as the scheduling constraints inside a node depend on its calling context. The underlying problem is not specific to this application and is generalized to a broader class of problems. We show how it can be solved optimally with an exponential algorithm or with a heuristic at the cost of an increased generated code size.

Several questions left open are discussed in Section 6. First, we use dynamically allocated arrays but we do not have a satisfactory way to deallocate them. The restrictive nature of dataflow synchronous languages hints we could do better than a full-fledge garbage collector. Second, our modular scheduling algorithm is limited to a single dimension (time) but we think it could be adapted into a multidimensional version. Then we could use the multidimensional scheduling algorithm to implement loops. Last, our algorithms still have a few limitations that should be lifted.

Parts of this internship report are being adapted in a paper to be submitted at the PLDI'15 conference.

1 Background

1.1 Dataflow synchronous languages

Dataflow synchronous languages [2] like Lustre [11], Signal [3] or Scade have been designed to program critical command systems. Such systems must interact with their environment and answer to external stimuli within a given time constraint. A violation of this time limit could result in disastrous consequences. Indeed dataflow synchronous languages are found in critical systems such as the autopilot of planes.

Time is too hard to reason about. Thus it is discretized into reactions. At each reaction, the program computes its new outputs from its inputs and its previous states. Within a single instant, everything happens as if computations where instantaneous. The environment can only see changes at the end of a reaction. Reactions happen at a fixed rate, chosen so that computations are all complete before the next reaction starts. This abstraction is commonly used in circuit design, where time is divided into clock cycles, and where wires have a stable value at the end of a cycle.

A dataflow synchronous program describes the computations within a single reaction. Then, this program is compiled into a step function which is called at each reaction. The step function computes the new value of variables from the inputs and the previous values of variables.

Each variable is defined by an equation and the compiler is responsible for ordering equations so that each variable is computed after all the variables it depends on are available. This way, the programmer only cares about the value of variables and not about the evaluation order. It also allows to write feedback loops on functions: a caller might feed an output of a node to one of its input. The main motivation for having feedback loops is to allow closed-loop control for embedded systems [5]. Moreover, they allow the implementation of registers [6] and the use of recursion by spreading it over time.

Not every equation is computed at each reaction. Instead, each variable is assigned a clock and is only defined when the clock is enabled (for example when a boolean condition is satisfied). This approach is similar to the clock gating technique used in circuit design.

In order to ensure the Worst Case Execution Time (WCET) of reactions can be computed, dataflow synchronous languages must follow some restrictions. Programmers cannot write loops or use recursion, at least in their common form. This way, only clocks can make the execution time vary and the WCET can be easily computed.

1.2 The base language

During this internship, we have used a simplified version of the Lustre programming language. Variables are either of scalar type `int` or `bool` or of array type `int[size]` or `bool[size]`. The role of functions is played by *nodes*. They can be thought of as stateful synchronous functions.

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

We restrict ourselves to a single clock. It means every equation is computed at each reaction. The semantics is purely functional: no side effects are visible and a variable has only a single value during a given reaction. The order of equations does not matter as the compiler reorders them. The size of arrays must be known statically. The following constructions are available:

```
(* Basic arithmetic operators *)
b: int = 1 * a + 2 - a
a: int = 4 / -2

(* Basic boolean operators *)
c: bool = not false
d: bool = (c || false) && true

(* Basic comparison operators *)
e: bool = a < b || b > a
f: bool = a == b || a != b
g: bool = a <= b || b >= a
```

```
(* Conditionals *)
h: int = if c then a else b

(* Fills an array with a *)
I: int[10] = a^10

(* Reads an array *)
j: int = I[b]

(* Copies the array I and assigns
 * 4 to the cell 2 *)
K: int[10] = I[2] <- 4
```

Feedback loops are allowed: an input might depend on the output of the same function. The value of variables at the previous reaction can be accessed using the **pre** operator. `->` is used to initialize variables at the first reaction, when **pre** `a` is not defined because `a` does not have a previous value.

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

```
(* 0 at the first reaction, then
 * the previous value of a + 1.
 * Computes 0, 1, 2, 3, ...*)
a: int = 0 -> pre a + 1
```

2 Motivation and previous work

The purely functional semantics of dataflow synchronous languages makes them easier to verify but might be a source of problems when performance is needed. Indeed, variables are constant and side effects are forbidden. Arrays (or actually any data structure) cannot be modified in-place: a full copy of the array has to be issued at each update.

A common way to circumvent this limitation is to rely on external functions, usually written in C, to handle the performance-critical parts of the application. This approach totally breaks the properties of the languages and is not satisfactory for critical systems.

This problem is not limited to dataflow synchronous languages and occurs whenever purely functional data structures are used. Several approaches to allow in-place updates while keeping a functional semantics have been explored both in the context of regular functional languages and of dataflow synchronous languages.

2.1 In-place updates in functional languages

A first idea to improve performance of arrays in purely functional languages is to use persistent data structures [8]. Instead of using a fresh copy of the data structure at each update, only the modifications are recorded. While this solution removes the cost of copying, it still incurs an overhead since array accesses must look for the possible modifications.

Another idea is to use destructive updates. When a data structure is updated, it is consumed and cannot be used any more. This way, the memory can be reused for the updated version and no copy is needed. If the type system ensures no data structure is used after it has been updated, every update can happen in-place. This solution can also be implemented as an optimization that detects when a data structure is not used

after an update. This does not require to change the type system, but there is no guarantee every update will be in-place.

2.2 Arrays in dataflow synchronous languages

Destructive updates have already been studied in the context of dataflow synchronous languages. An algorithm that optimizes some copies into destructive updates in the LabVIEW dataflow synchronous language is introduced in [1], However this algorithm does not handle inter-procedural optimizations and it does not remove every copy.

Another algorithm achieving a similar goal is proposed in [9], but it suffers from similar drawbacks as [1] unless the programmer manually annotates the program to tag arrays that should be stored at the same location and updated in-place.

This annotation based approach, when strictly enforced with a type system, allows inter-procedural copy-avoiding and gives a guarantee to the programmer that no copy will occur, but it requires manual intervention and is overly restrictive.

3 Destructive arrays

We propose a method to ensure every definition of an array element is compiled into a destructive update. Unlike the existing propositions introduced in Section 2.2, it does not require annotations, it handles inter-procedural data-flow, uses dynamically allocated arrays and it does not alter the genericity of nodes.

If an update cannot be compiled into a destructive update, the program is rejected. This way, the programmer has the guarantee no hidden copy will affect performance. If a program requires a copy to be accepted, the programmer can explicitly ask for it using the `copy` keyword.

Previous propositions rely on statically allocated arrays variables. Each array variable has a fixed location assigned during compilation. Variables might share location, but this aliasing cannot be input dependent. This limitation is a source of unnecessary copies. In the following example, `C` can either alias with `A` or `B` but not both. The other array would have to be copied. To solve this problem, we use dynamic allocation.

```
(* With static allocation, C cannot alias with both A and B *)
C: int[10] = if x then A else B
```

3.1 Scheduling constraints

As explained in Section 2.1, functional languages usually use the type system to ensure every update can be compiled as a destructive update. Instead, we exploit the degrees of freedom offered by a dataflow language, translating its equational definitions into scheduling constraints.

A dependency is added between each use of an array and its consumption. This way, no array is accessed after it has been consumed. In practice, it boils down to adding dependencies from read and writes to writes on aliasing variables. This is not trivial as aliasing might depend on inputs and might occur across reactions.

```
(* Allocates a new array *)
A: int[10] = 0^10

(* Adds a dependency from d to C
 * d is computed before C *)
C: int[10] = A[8] <- 42
d: int = A[3]

(* Adds a dependency from g to C
 * depending on x *)
E: int[10] = copy A
F: int[10] = if x then A else E
g: int = F[6]
```

```
(* Adds a dependency across
 * reactions from pre C to e *)
e: int = (pre A)[3]

(* Creates a dependency cycle as A
 * is written to twice, so C and F
 * depend on each other *)
F: int[10] = A[3] <- 4

(* Creates a dependency cycle as A
 * is consumed when C is created *)
g: int = C[A[5]]
```

The use of scheduling constraints instead of a type system leverages the flexibility of the implicitly scheduled semantics of dataflow synchronous languages to automatically adapt the ordering of equations for destructive updates. The compiler might even shift a computation to the previous or the next reaction if needed. This flexibility greatly improves the genericity of nodes as they can adapt to the aliasing and the scheduling constraints of their calling context.

3.2 Modular compilation

The compiler first decomposes each equation into basic operations to add more flexibility to the scheduling. Then it schedules equations to respect dependencies.

```
(* 1: Initial equations *)
b: int = a*a + 5
a: int = 42

(* 2: Decomposed equations *)
b_0: int = a*a
b_1: int = b_0 + 5
a: int = 42
```

```
(* 3: Scheduled equations *)
a: int = 42
b_0: int = a*a
b_1: int = b_0 + 5
```

It requires the compiler to know all the dependencies before scheduling a node. However, dependencies might depend on the calling context. They can come either from a feedback loop or from aliasing in the node arguments. Thus, all nodes must be inlined before the program is compiled. This is not a satisfactory solution since it does not allow separate compilation. The full program has to be recompiled at each update and the generated code might grow exponentially with regard to the input code.

A solution, used in SCADE [4], is to forbid feedback loops unless the programmer manually ask for inlining. However, it either adds artificial constraints on the system or keeps the problem of inlining. A modular scheduling algorithm that does not add any constraint while still avoiding inlining is presented in Section 4.

4 Generalization and solutions to the scheduling problem

In this section, we generalize the modular scheduling problem presented in Section 3.2 to a broader class of systems that we call modular systems of equations. Then, we show a full inlining can be avoided by finding a

grey-boxing of nodes. We formalize this notion of grey-boxing and give an optimal algorithm and a heuristic to find good grey-boxings.

4.1 Modular systems of equations

As for dataflow synchronous languages, modular systems of equations are organized in nodes. Each node has a set of variables V, a set of inputs I and a set of outputs O. Each variable $a_i \in V_{\setminus I}$ is defined by an equation of the form:

$$\forall t \geq 0 : \; a_i(t) = f_i\left(a_{i_0}(t - w_0^i),\, a_{i_1}(t - w_1^i),\, \ldots,\, a_{i_{k-1}}(t - w_{k-1}^i)\right) \tag{1}$$

Where $a_{i_j} \in V$, $w_j^i \in \mathbb{Z}$ for $j \in [\![0, k-1]\!]$. The values of $a_i(t)$ for $t < 0$ are considered given. Additionally, some variables of $V \setminus I$ might be defined by calls to another node instead of a regular equation. A call (N', I', O') is defined by the name of the called node N', a list of variables I' to define the inputs of N' and a list of variables O' that will be defined by the outputs of N'. The goal is to find an evaluation order for the equations that respects the dependencies to get values of the $a_i(t)$.

Modular systems of equations are a generalization of dataflow synchronous languages without clocks. Indeed, the equation `a = pre b` can be seen as $a(t) = b(t-1)$ and the other operators can be seen as equations of the form $a_i(t) = f_i\left(a_{i_0}(t), \ldots, a_{i_k}(t)\right)$.

4.2 Dependency relations and dependency graph

In (1), w_j^i does not depend on t. Thus, dependencies among equations can be represented independently of t. It is achieved through the dependency relation $\prec$. For $a, b \in V$, and $w \in \mathbb{Z}$:

$$b \overset{w}{\prec} a \iff \forall t \geq 0 : \; a(t) \text{ depends on } b(t-w) \iff \exists t \geq 0 : \; a(t) \text{ depends on } b(t-w)$$

The dependency relation is transitive:

$$a \overset{w_{ab}}{\prec} b \wedge b \overset{w_{bc}}{\prec} c \implies a \overset{w_{ab}+w_{bc}}{\prec} c \tag{2}$$

A node N is said to be *causally correct* (noted $causal(N)$) if there is an ordering of equations that respects dependencies.

Proposition 1. *A node is causally correct iff there is no dependency cycle:*

$$causal(N) \iff \forall a \in V, w \in \mathbb{Z} : \; a \overset{w}{\prec} a \implies w > 0$$

Proof. If $\forall a \in V, w \in \mathbb{Z} : \; a \overset{w}{\prec} a \implies w > 0$, Section 4.3 gives an algorithm to find a valid ordering. Otherwise, $\exists a \in V$ and $w \leq 0$ such that $\forall t \geq 0 : \; a(t)$ depends on $a(t-w)$. As $\forall n \in \mathbb{N} : \; n \cdot w \geq 0$, $a(n \cdot w)$ depends on $a((n+1) \cdot w)$ for all $n \geq 0$. Thus, there is an infinite dependency chain starting from $a(0)$ and no ordering of equations will ever be valid. □

In the following, we assume:

$$\forall a \in V : \; a \overset{1}{\prec} a \tag{3}$$

Adding this dependency does not alter the correctness of a node since it cannot create new dependency cycles. It allows us to only compute the smallest weigh on dependencies since (3) combined with (2) gives us:

$$\forall a, b \in V, w \in \mathbb{Z} : \; a \overset{w}{\prec} b \implies \forall w' > w : \; a \overset{w'}{\prec} b$$

Additionally, we define the *depends or equal* relation as:

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

This relation is easier to reason about in some cases and when a node is causally correct, the dependency relation $\prec$ can be computed from it:

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

The dependency relation can be easily represented and computed on a weighted directed graph G. The vertices of the graph are the variables V. There is a non-empty path from a to b with weight w in G iff $\forall w' > w : a \overset{w'}{\prec} b$. The edges of G are defined as follows:

- If $b(t) = f(\ldots, a(t-w), \ldots)$, an edge $a \xrightarrow{w} b$ is added to G.
- If (N', I', O') is a call, $a \in I'$ corresponds to the input i of N', $b \in O'$ corresponds to the output o of N' and $i \overset{w}{\preccurlyeq} o$ in N' with w minimal, then an edge $a \xrightarrow{w} b$ is added to G.

Then distances between vertices are computed using a shortest path algorithm to gets the dependency relation. Dependency cycles can be detected by looking for cycles with a weight ≤ 0.

4.3 Scheduling of equations

As for synchronous languages, we look for a scheduling of equations in the form of a step function that computes a single reaction and that can be iterated to get the values of variables for all values of t. As shown by the algorithm given below, such a schedule can always be found as long as the node is causally correct. To formalize this idea, we introduce the notion of static schedule.

Definition 1 (Static schedule)**.** A static schedule 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:

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

Non-modular scheduling algorithm

Let N be a causally correct node. Then a static schedule for N can be found using the following algorithm.

First, a retiming is found by iteratively removing dependencies with negative weight. Initially, $\forall a \in V : r(a) = 0$. Then, while:

$$\exists a, b \in V : a \overset{w}{\prec} b \wedge r(b) - r(a) + w < 0$$

such that w is minimal, r is updated with $r(b) := r(a) - w$. Thus, in the end, $\forall a, b \in V : a \overset{w}{\prec} b \implies r(b) - r(a) + w \geq 0$. To efficiently compute the retiming, we can restrict ourselves to the edges of the dependency graph when looking for negative dependencies.

Proposition 2. *If there is no dependency cycle, the retiming algorithm terminates.*

Once a retiming is found, we must order dependencies within a single reaction. To do this, we look at the dependency graph G of N and only consider the edges with a null weight. When $a \xrightarrow{n} b \in G$ such that $r(b) - r(a) + n = 0$, an edge $a \longrightarrow b$ is added to G'. Then, the equations are ordered using a topological sort on G'. It is always possible as a cycle in G' can only come from a cycle with null weight in G which is impossible since N is causally correct.

Proposition 3. *The algorithm produce a correct static schedule.*

4.4 Grey-boxing

The algorithm introduced inSection 4.3 does not handle calls and requires every node to be inlined. As explained in Section 3.2, it is problematic but compiling a node in a single step function is just impossible since the calling context might add some dependencies between outputs and inputs.

The idea of grey-boxing is halfway between the full inlining (white-boxing, the caller sees the full implementation of the node) and the compilation into a single step function (black-boxing, the caller sees only the step function). The node is compiled into a few atomic sub-nodes. Each sub-node is a group of equations that can always be executed together, independently of the context. Sub-nodes can be scheduled and retimed with regard to each other, but the schedule inside a sub-node remains the same. For example, on Figure 1, 3 sub-nodes can be created.

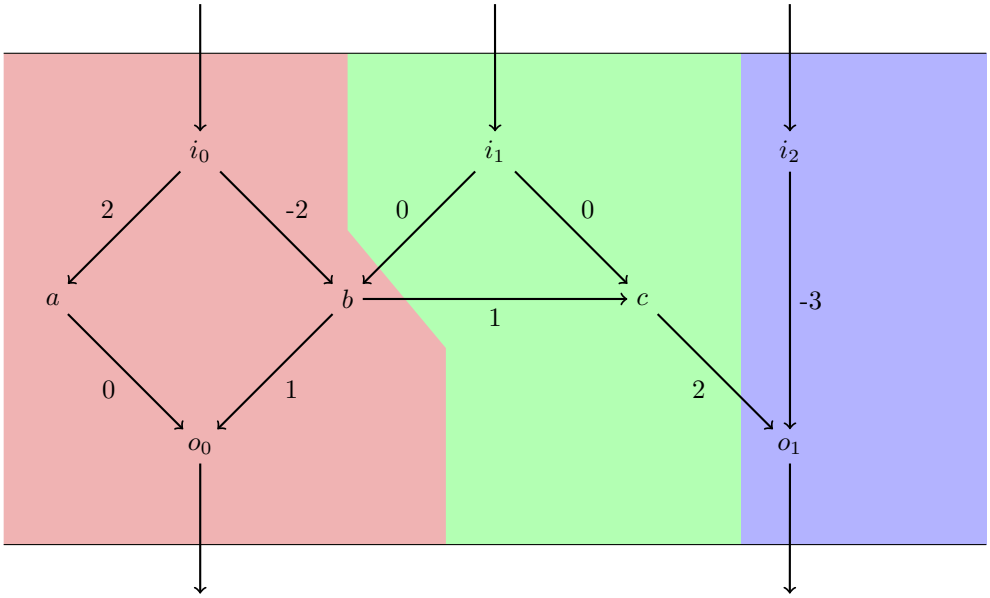

Figure 1: Example of grey-boxing on the dependency graph

To stay as far as possible from the full inlining, the number of sub-nodes should be minimal. The problem of finding a minimal partitioning has been studied in [13] without retiming and proven to be NP-complete. However, Pouzet and Raymond have revisited this problem, formalized it and given a heuristic that can compute an optimal solution in most cases [14]. The algorithms and the formalization we introduce next are based on their solutions but takes retiming into account while they do not.

4.5 Grey-boxing formalization

4.5.1 Weighted preorder

The idea to formalize grey-boxing is to use an over-constraint version of the dependency relation to form the groups of equations to compute together. When two equations depend on each other, they must be computed in the same atomic sub-node. For this purpose, we introduce the notions of weighted preorder and weighted equivalence.

Definition 2 (Weighted preorder)**.** 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 3 (Weighted equivalence)**.** 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 4. *A valid weighted preorder induces a weighted equivalence:*

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

4.5.2 Grey-boxing induced by a weighted preorder

A valid preorder respecting dependencies (i.e., $a \overset{n}{\prec} b \implies a \overset{n}{\precsim} b$) induces a grey-boxing. Each equivalence class represents a different atomic sub-node. The dependencies between sub-nodes are given by quotienting $\precsim$ by $\simeq$.

Let $\precsim$ be a valid weighted preorder that respects dependencies and $\simeq$ its associated weighted equivalence. Let $X_0, \ldots, X_{k-1}$ be the equivalence classes of $\simeq$ and thus the different sub-nodes. Let $(r_i, <_i)$ be the static schedule associated with each X_i and $\prec_X$ the dependency relation between sub-nodes. $\prec_X$ is used by the calling context to know how to order the atomic sub-nodes of the grey-boxing with regard to each other.

The retiming inside a sub-node can only be defined up to an additive constant. Indeed, the retiming of the sub-node itself in the instantiated context is unknown. Thus, we 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, $<_i$ as in the scheduling algorithm introduced in Section 4.3.

Remark 1. *The retiming r_i does not depend on the choice of a.*

Once each retiming is defined, we can give the dependency relation between sub-nodes. It can be thought of as some kind quotienting of $\precsim$ by $\simeq$. As $\prec_X$ is causally correct (because negative cycles are forbidden since $\precsim$ is valid and that equations on a null-weight cycle end up in the same equivalence class), it can be recovered from $\preccurlyeq_X$. Thus, we only define $\preccurlyeq_X$:

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

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

Remark 2. *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 grey boxing. Indeed, let $X_0 \ldots, X_{k-1}$ be a partitioning of equations, r_i the retiming insides each sub-node and $\preccurlyeq_X$ the depends-or-equal 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 grey-boxing. It means that by looking at valid weighted preorder we are looking at every possible grey-boxing.

4.5.3 Static partitioning

To ensure the grey-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 5, a static partitioning is exactly a preorder that respects dependencies and does not add constraints on possible calling contexts.

Definition 4 (Static partitionning)**.** 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 5 (Static partitionings are the preorders that do not reject any valid context)**.** *Let N be a node and $\precsim$ be a valid preorder respecting the dependencies of N. For every calling context C, let $G(C)$ be the node C where the call to N has be replaced by the grey-boxing induced by $\precsim$. Then:*

$$\precsim \text{ is a static partitioning} \iff \forall C : causal(C) \implies causal(G(C)) \tag{7}$$

Remark 3. *The depends-or-equal relation $\preccurlyeq$ is a static partitioning. It corresponds to the grey-boxing where each equation is a separate box, or in other word the full inlining of equations in the calling context.*

4.5.4 Compatibility

To give hints on how to group equations for a grey-boxing, we define the compatibility relation:

Definition 5 (Compatibility)**.** 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 6, the compatibility is a necessary condition to group equations together. However, it is not sufficient to prove a grey-boxing is correct. Static partitionings should be used instead for this purpose.

Proposition 6. *If $\overset{n}{\precsim}$ is a static partitioning and $\overset{n}{\simeq}$ its associated weighted equivalence, then:*

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

4.6 Heuristic

When two equations depend on the same inputs or the same outputs, then they can be grouped together. We use this property to create a heuristic that builds a static partitioning optimal on inputs and outputs. It is achieved by starting from the depends or equal relation $\preccurlyeq$ (which is a static partitioning according to Remark 3) and successively adding constraints to the preorder using information given by dependencies between inputs, outputs and equations.

The code that does not depend on any input or that does not have any output depending on it (the dead code) does not behave well with our heuristic. Indeed it lacks a bound on how early or how late it can be scheduled and sometimes ends up with an infinite retiming. Such equations are detected and handled separately. The dead code is removed and the code that does not depend on any input is gathered into an additional sub-node of the grey-boxing. Inside the additional node, equations are scheduled using the algorithm introduced in Section 4.3. In the following we assume such code has been taken care of and that every equation depends on at least one input and has an output depending on it.

The constraints are added to the preorder using the input and output saturation functions.

Definition 6 (Input and output saturations)**.** Given a static partitioning $\precsim$, we define its input function $\mathcal{I}_n^{\precsim}$ (resp. output function $\mathcal{O}_n^{\precsim}$) and its input saturation $\precsim_{\mathcal{I}}$ (resp. output saturation $\precsim_{\mathcal{O}}$):

$$\begin{array}{rcl}
\mathcal{I}_m^{\precsim}(x) & = & \left\{ (i,n) \in I \times \mathbb{Z} \;\middle|\; i \overset{n+m}{\precsim} x \right\} \\
x \overset{m}{\precsim_{\mathcal{I}}} y & \iff & \mathcal{I}_0^{\precsim}(x) \subseteq \mathcal{I}_m^{\precsim}(y) \\
\mathcal{O}_m^{\precsim}(x) & = & \left\{ (o,n) \in O \times \mathbb{Z} \;\middle|\; x \overset{n+m}{\precsim} o \right\} \\
x \overset{m}{\precsim_{\mathcal{O}}} y & \iff & \mathcal{O}_m^{\precsim}(x) \supseteq \mathcal{O}_0^{\precsim}(y)
\end{array}$$

Proposition 7 shows us the input and the output saturation only add constraints to the initial preorder and remain static partitionings. Moreover, they are optimal respectively on outputs and inputs as their induced equivalence relations meet the compatibility relation on these subsets. Thus, any pair of inputs (resp. outputs) that could be grouped together are put in the same sub-node by the output saturation (resp. input saturation).

Proposition 7. *$\precsim_{\mathcal{I}}$ 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 $i \overset{n}{\precsim} i \implies i \overset{n+1}{\precsim} x$, $\mathcal{I}_m^{\precsim}(x)$ 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 $\mathcal{O}_m^{\precsim}(x)$.

Definition 7 (Input/output saturation)**.** Let $\precsim$ be a static partitioning. Its input/output saturation $\precsim_{\mathcal{IO}}$ 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 8 shows it is also optimal on input-output pairs.

Proposition 8. *The input/output-saturation meets the compatibility on input-output pairs:*

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

In the end, we have a 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 (see Section 4.4) 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. Thus, it is not a problem to have only an approximation of the optimal solution.

4.7 Exact algorithm

To find 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 retiming taken into account (see Section 4.4).

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 a and b are comparable:

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

- d_{ab}, $a, b \in V$: an integer which has a meaning only when $c_{ab} = true$, 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 ensured by the following formula:

$$\bigwedge_{a \in V} c_{aa} \wedge w_{aa} \leq 0 \bigwedge_{a,\, b,\, c \in V} (c_{ab} \wedge c_{bc}) \implies (c_{ac} \wedge w_{ac} \leq w_{ab} + w_{bc}) \tag{8}$$

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

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

In order to count the number of classes, an additional integer variable x_a is added $\forall a \in V$. Each value of x_a represents a different class. The relations between the x_a and $\simeq$ as well as the limitation on the number of classes are encoded in:

$$\bigwedge_{a \in V} 0 \leq x_a < k \bigwedge_{a,\, b \in V} x_a = x_b \implies (c_{ab} \wedge c_{ba} \wedge w_{ab} = -w_{ba}) \tag{10}$$

An assignment of variables that satisfy the conjunction of (8), (9) and (10) gives 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, an off-the-shelf SMT solver can be used.

5 Modular compilation of destructive arrays

In this section, we come back to the initial problem of modular compilation of destructive updates raised in Section 3.2. First, we show how an approximation of the aliasing relation can be computed. Then, we show how this modular compilation problem can be encoded in the formalism of Section 4.

5.1 The aliasing relation

Two variables alias if they are the same array from a functional point of view. The information is needed to find when to add dependencies between reads and writes. The aliasing relation $\sim$ is the smallest equivalence relation respecting the constraints presented in Table 1

Equation	Condition	Aliasing
`a = b`		$a(t) \sim b(t)$
`a = if b then c else d`	$b(t) = true$	$a(t) \sim c(t)$
`a = if b then c else d`	$b(t) = false$	$a(t) \sim d(t)$
`a = pre b`		$a(t) \sim b(t-1)$
`a = b -> c`	$t = 0$	$a(0) \sim b(0)$
`a = b -> c`	$t > 0$	$a(t) \sim c(t)$

Table 1: Aliasing relation

The problem with the relation presented in Table 1 is that it deals with the values of variables at a given instant while we are looking for a schedule independent of t. Instead, we only consider the aliasing distance relation $\overset{n \in \mathbb{Z}}{\sim}$ defined as:

$$a \overset{n}{\sim} b \iff \exists t :\ a(n) \sim b(n+t)$$

This second relation represents all the possible alias encountered when t ranges over $\mathbb{N}$. While $\sim$ is an equivalence relation, $\overset{n \in \mathbb{Z}}{\sim}$ is not even transitive (one might have $a \overset{n}{\sim} b \wedge b \overset{m}{\sim} c$ but $\neg \left(a \overset{n+m}{\sim} c\right)$).

Even with the time abstracted away, the aliasing is impossible to compute. Indeed, it would require to know if two arbitrary arithmetic expressions are equivalent or not, which is undecidable. Instead, we use an

over-approximation. We have restricted ourselves to a very conservative one as aliasing analysis was not the main subject of this internship.

The approximation is based on the notion of ancestor. The ancestor relation keeps track of the origin of values assigned to variables. For example, if `a = b`, `b` is an ancestor of `a` and if `a = if b then c else d`, both `c` and `d` are ancestors of `a`. As for the dependency relation, the ancestor relation can be represented with a graph. There is a path from a to b with weight n when $a(t)$ is an ancestor of $a(t+n)$. Formally, the vertices are the variables and the edges are defined as in Table 2.

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 2: Ancestor graph edges

From this graph, the approximation of the aliasing distance can be computed. Two variable alias if they have a common ancestor:

$$a \overset{m-n}{\sim} b \iff s \xrightarrow{n}{}^{*} a \wedge s \xrightarrow{m}{}^{*} b$$

The aliasing distance is only used to add dependencies between array accesses. Since only the smallest dependency distance is needed (see Section 4.2), only the smallest aliasing distance between each pair of equations must be computed. Moreover, it is sufficient to consider only the sources of new arrays when looking for a common ancestor. This way, we only compute the minimum and the maximum distance between sources and other variables to compute the aliasing distance. If s is a source, a a variable at the maximal distance m of s and b a variable at the minimal distance n of s, then $a \overset{m-n}{\sim} b$.

For inter-procedural aliasing analysis, the compiler keeps a small summary of the ancestor graph of each node. It has the same behavior than the full ancestor graph with regard to the aliasing distance but only contains input and output variables. Let G_a be the ancestor graph of a node, I the inputs, O the outputs and S the sources of new arrays of the node. The corresponding summary node G'_a is defined on vertices $I \cup O$ by:

- If $i \in I$, $o \in O$, the minimal distance from i to o in G_a is m_0, and the maximal distance m_1 then G'_a has the edges $i \xrightarrow{m_0} o$ and $i \xrightarrow{m_1} o$.
- If o_0, $o_1 \in O$ and $\exists s \in S \backslash I$ such that the minimal distance from s to o_0 in G_a is m_0 and the maximal distance from s to o_1 in G_a is m_1, then G'_a has the edge $o_1 \xrightarrow{m_0-m_1} o_0$.

During the aliasing analysis, we also check that no array is written to twice. This way, only dependencies from reads to write needs to be taken into account by the rest of the compilation flow as dependencies between two writes will have already made the compiler complain. The aliasing summary keeps track of inputs and outputs that have been written to handle inter-procedural write-write cycles.

5.2 Encoding as a modular scheduling problem

To find a modular schedule for a node, we encode its dependencies in a modular system of equations and use the grey-boxing technique introduced in Section 4. The translation is straightforward except for dependencies added by destructive updates. Indeed, nodes are already of the form presented in Section 4.1.

Constraints on array accesses are inferred using the aliasing distance over-approximation. If a and b are defined by `a = c[.]` and `b = d[.] <- .`, and $c \overset{n}{\sim} d$, then $b(t)$ depends on $a(t-n)$. The problem is that the calling context can add aliasing between inputs or access the same arrays than the body of the functions. Thus, some dependencies are not known until the node is instantiated.

This problem is very similar to the modular scheduling problem tackled in Section 4. In both cases, dependencies are added by the calling context and change the dependencies between equations inside the node. For modular systems of equations, dependencies added by the context only involves inputs (for inbound dependencies) and outputs (for outbound dependencies). Here, dependencies all go from a read to a write. Thus, we encode reads as outputs and writes as inputs. Then, the calling context can add a dependency that was unknown when the called node was compiled as if it was interacting with the node inputs and outputs and the modular scheduling algorithms of Section 4 can be used. The aliasing summary is enhanced to get the aliasing between the accesses exposed as inputs or outputs and the actual inputs and outputs.

For example, in Figure 2, the regular dependencies between equations (in black) are statically known, but the aliasing between `A` and `B` is unknown at the time `foo` is compiled. Thus, the read from `A` and the write to `B` are added as inputs and outputs. When `foo` is instantiated, the calling context knows `A` and `B` alias. It can add a dependency from the read to the write (in dashed red). Moreover, the context reads into an array aliasing with `B`, thus an additional dependency must be added from the context to the write to `B`.

```
(* Node to compile modularly,
 * in grey on the figure *)
node foo(A: int[10], B: int[10])
    = (c: int) {
  D: int[10] = B[0] <- 1
  E: int[32] = 4^32
  f: int = A[2] + D[2] + E[2]
  c: int = f
}

(* Calling context *)
X: int[10] = 42^10
y: int = foo(X, X)
z: int = X[9]
```

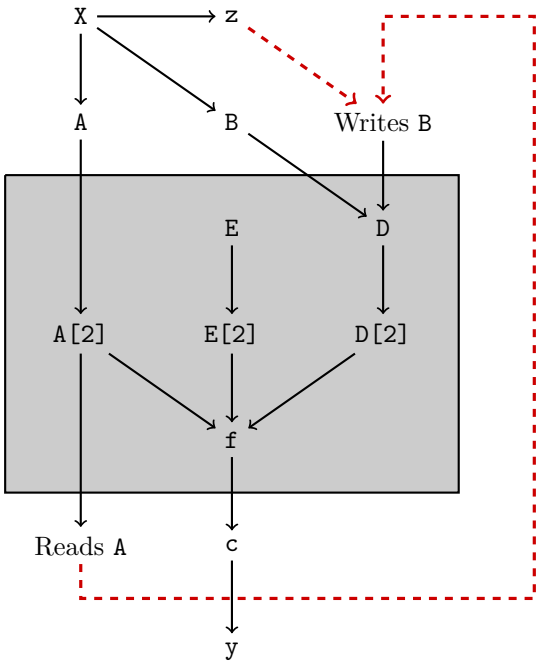

Figure 2: Example of encoding of reads and writes into inputs and outputs

Actually, not every read or write can be a source of new dependencies when the node is instantiated. Moreover, not every pattern of dependency is possible.

- Reads and writes must occur on arrays aliasing with an input or an output of the node. Otherwise, all dependencies involving the read or the write will be known before instantiation. Indeed, the caller cannot create additional aliasing or perform read/write operations with arrays it does not have access to. Thus, only array aliasing with an input or an output are considered when adding reads and writes as outputs and inputs. As a variable can only be written to once, the number of writes exposed as inputs is limited by the initial number of inputs and outputs.

- Arrays can only be written to once. If a variable only alias with arrays that have been written to, it cannot be written to. Thus, a read from such a variable cannot produce any additional dependency and is not considered as a source of dependency.
- All the reads from the same array creates a dependency together when a write to an aliasing array is used. Since a variable might alias with multiple arrays, it is hard to take this into account. We solve this problem by duplicating a read for each source of new array it alias with. Then, we group reads from the same sources. As the only sources in the aliasing summary are the inputs and outputs, it also ensures the number of reads exposed as outputs does not exceed the initial number of inputs and outputs. While this optimization decreases the number of reads and handles reads from the same array, it can add constraints on the possible grey-boxing in some cases.
- When an array is created by a node and read from or written to by its calling context, the array first has to *escape* the node: the access can only happen after the array is outputted by the node. It also restricts the possible dependencies.

We still have not found an exact solution to take the two last points into account. Thus, our encoding of the problem is too conservative. This results is a sub-optimal grey-boxing but does not alter the correctness and does not change the set of accepted callers.

While sub-optimal, this encoding gives us a way to handle unknown dependencies in a modular way and reuse the algorithms presented in Section 4. Moreover, the encoding only increases the apparent number of inputs and outputs in a linear way.

6 Future work

In this section, we introduce a few possible future directions of research. First, we need a static garbage collector to deallocate our dynamically allocated arrays. Then, we could try to adapt our modular scheduling algorithm to handle multidimensional systems of equations. We could also look for a way to express recursion and loops in our base language. Last, some limitations of our current solution would need to be lifted.

6.1 Static garbage collector

As explained in Section 3, we use dynamic allocation instead of static allocation for arrays. This removes some copies that would otherwise be necessary while still allowing to bound the memory usage as there cannot be more arrays than array variables. However, we do not treat the problem of deallocation which is necessary to avoid memory leaks.

A full-fledged garbage collector would be cumbersome and would make it almost impossible to give a WCET for a reaction. However, some kind of static garbage collector embedded in the code could do the trick as long as it is predictable enough. For example, the compiler could insert code to detect which arrays are still alive at the end of a reaction. The compiler could also use the information provided by the aliasing analysis on reads and writes to know when arrays will not be accessed anymore and free them.

6.2 Multidimensional modular system of equations

The modular systems of equations presented in Section 4.1 are quite close to the Systems of Uniform Recurrence Equations (SURE) defined by Karp, Miller and Winograd [12]. The main differences reside in the fact that SURE do not have inputs and outputs and cannot be instantiated. Moreover, SURE iterate over a

multidimensional finite domain while modular systems of equations iterate over $\mathbb{N}$. Formally, each equation is of the form:

$$\forall t \in R:\ a_i(t) = f_i\left(a_{i_0}(t - w_0^i),\ a_{i_1}(t - w_1^i),\ \ldots,\ a_{i_{k-1}}(t - w_{k-1}^i)\right) \tag{11}$$

Where the iteration domain R is a finite subset of $\mathbb{Z}^n$ and $w_j^i \in \mathbb{Z}^n$ for $j \in [\![0,\ k-1]\!]$. The values of $a_i(t)$ for $t \notin R$ are considered given. As for modular systems of equations, we can define a dependency relation $\prec$, but with weights in $\mathbb{Z}^n$ instead of $\mathbb{Z}$. A SURE is causally correct if:

$$\forall a \in V, w \in \mathbb{Z}^n : a \overset{w}{\prec} a \implies w \neq 0^n$$

The scheduling of non-modular SURE has been extensively studied by Darte and Vivien in [7]. They show we can restrict ourselves to *affine schedules*. Such schedules can be seen as a generalization of the compilation to a step function used in dataflow languages.

By adding inputs and outputs to SURE, we could allow them to be instantiated as we did with modular systems of equations. It raises the same modular scheduling problem as described in Section 4. We think the same grey-boxing technique with the same formalization, the same heuristic and the same optimal algorithm but with multidimensional weights could achieve similar results.

Polyhedra could be used to efficiently represent all the dependencies between two equations. Indeed, with the schedules described by Darte and Vivien, if $a(t)$ is computed before $b(t - w_0), b(t - w_1), \ldots, b(t - w_k)$, then $a(t)$ is computed before $b(t - w)$ for any w in the convex hull of $w_0, \ldots, w_k$.

6.3 Loops and recursion

Synchronous languages usually do not feature recursion or loops. It is a problem when dealing with arrays where a computation must be repeated on each element. Guatto and Mandel show in [10] how to use burst of values to implement nested loops. Here we propose two alternative approaches.

First we could implement nested loops using a multidimensional version of our modular scheduling algorithm, as the one discussed above in Section 6.2. Each loop would add a new dimension to the equations of its body to represent the different iterations. With this method, the WCET of a reaction could still be easily computed as long as the number of iterations of each loop is known statically.

Second, we could implement recursion. For this, we need a way to compute a fix-point on the aliasing and on the dependency relation. However, we are confident this can be achieved. While recursion is more expressive than loops, we still need to figure how a WCET could be computed.

6.4 Limitations to be lifted

Our proposition to introduce destructive updates and our modular scheduling algorithm have a few limitations that should be lifted.

First, the base language we have used during this internship does not feature different clocks while conventional dataflow synchronous languages do. Thus, we should find a way to handle equations that are evaluated at a different pace.

Second, we use a very conservative aliasing analysis. An extensive analysis of the existing literature and some research would be needed to improve it.

Last, as explained in Section 5.2, the encoding of reads and writes as inputs and outputs is still perfectible. Indeed, the current encoding allows too much constrains to be added by the context. It results in a sub-optimal grey-boxing.

7 Conclusion

In this internship report, we have introduced a novel method to enforce destructive updates. While we focus here on arrays, our method could apply to any data structure. Our proposition uses the flexibility of the implicitly scheduled nature of dataflow synchronous languages to automatically adapt the evaluation order of equations depending on the calling context to allow destructive updates. It raises the question of how to modularly compile a node when the context can add dependencies between equations. We enhance an existing solution called grey-boxing to handle retiming and provide both an exact algorithm and a heuristic to find a grey-boxing. The sub-optimal solution found by the heuristic increases the generated code size but do not restrict any possible calling context and do not alter the correctness of nodes. Then, we show how destructive updates can be used with our modular scheduling algorithm by encoding array accesses as inputs and outputs. Once again, this encoding is sub-optimal, but it only increases the generated code size.

A few questions are left open. First, how to deallocate arrays. Second, how to handle loops or recursion. Then, how to use the grey-boxing in a multidimensional context, and last, how to lift the few limitations that remain on our destructive updates.

References

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

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

[3] Benveniste, A., Guernic, P. L., and Jacquemot, C. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*.

[4] Biernacki, D., Colaço, J.-L., Hamon, G., and Pouzet, M. Clock-directed modular code generation for synchronous data-flow languages. In *Conference on Languages, Compilers, and Tools for Embedded Systems* (Tucson, AZ, USA, 2008), LCTES '08.

[5] Caspi, P., Pilaud, D., Halbwachs, N., and Plaice, J. A. Lustre: A declarative language for real-time programming. In *Symposium on Principles of Programming Languages* (Munich, West Germany, 1987), POPL'87.

[6] Colaço, J.-L., Pagano, B., and Pouzet, M. A conservative extension of synchronous data-flow with state machines. In *ACM International Conference on Embedded Software* (Jersey city, New Jersey, USA, 2005), EMSOFT'05.

[7] Darte, A., and Vivien, F. Revisiting the decomposition of karp, miller and winograd. In *International Conference on Application Specific Array Processors* (Strasbourg, France, 1995), ASAP'95.

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

[9] 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 *International Conference on Languages, Compilers, Tools and Theory for Embedded Systems* (Beijing, China, 2012), LCTES'12.

[10] Guatto, A., and Mandel, L. Bursty kahn networks and integer clocks. In *JournÃ©es Francophones des Langages Applicatifs* (FrÃ©jus, France, 2014), JFLA'14.

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

[12] Karp, R. M., Miller, R. E., and Winograd, S. The organization of computations for uniform recurrence equations. *J. ACM 14*, 3 (1967).

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

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

Appendix A Proofs

A.1 Proof of Proposition 2

Proof. Let N be a node and V its variables. Lets assume the scheduling algorithm presented in Section 4.3 does not terminates. Then there exist an infinite sequence of pair of variables $(a_i, b_i)_{i\in\mathbb{N}}$, a sequence of weight $(w_i)_{i\in\mathbb{N}}$ and a sequence of retiming function $(r_i)_{i\in\mathbb{N}}$ such that:

$$a_i \overset{w_i}{\prec} b_i \wedge r_i(b_i) - r_i(a_i) + w_i < 0$$

and r_{i+1} is r_i updated with $r_{i+1}(b_i) = r_i(a_i) - w_i$. An occurrence of a is a $i \in \mathbb{N}$ such that a_i. For a given $i \in \mathbb{N}$, a predecessor is a $j < i$ such that $b_j = a_i$. Between two consecutive occurrence i_0 and i_1 of the same pair of variable (a, b) there is always a predecessor of i_1. Indeed, otherwise $r(a)$ would be left unchanged, the w would be the same as the pair of variable is the same and since r can only increase we would have:

$$r_{i_1}(b_{i_1}) - r_{i_1}(a_{i_1}) + w_{i_1} > r_{i_0}(b_{i_0}) - r_{i_1}(a_{i_1}) + w_{i_0} = 0$$

In particular, if i_0 is the last predecessor of i_1, then $r_{i_1}(a_{i_1}) = r_{i_0}(a_{i_0}) - w_{i_0}$. As the sequence is infinite, there is a pair of variables (a, b) that appear infinitely often. Thus is has an infinity of last predecessors. Among this last predecessors, a pair must also appear infinitely often and so on. Thus we can find a pair with a chain of last predecessor as long as we want. As there is finitely many variables, we can find a last predecessor cycle, i.e. $i_0 < i_1 < \cdots < i_k$ such that:

$$b_k = a_0 \wedge \forall j \in [\![0,\, k-1]\!] :\ i_j \text{ is the last predecessor of } i_{j+1}$$

Thus $r_{i_{j+1}}(a_{i_{j+1}}) = r_{i_j}(a_{i_j}) - w_{i_j}$ thus $r_{i_k}(a_{i_k}) = r_{i_0}(a_{i_0}) - \sum_{j=0}^{k-1} w_{i_j}$. But since $r_{i_k}(b_{i_k}) - r_{i_k}(a_{i_k}) + w_{i_k} < 0$ and $b_{i_k} = a_{i_0}$, $r_{i_k}(a_{i_0}) - r_{i_0}(a_{i_0}) + \sum_{j=0}^{k} w_{i_j} < 0$ and as r can only increase over time, $r_{i_k}(a_{i_0}) - r_{i_0}(a_{i_0}) > 0$ so $\sum_{j=0}^{k} w_{i_j} < 0$. Thus there is a negative dependency cycle and the node N is not causally correct. □

A.2 Proof of Proposition 3

Proof. Let N be a node, V its variables and G its dependency graph. Let $(r, <)$ a static schedule of N obtained with the algorithm presented in Section 4.3. Let $a, b \in V$ and $w \in \mathbb{N}$ such that $a \overset{w}{\prec} b$. Then, $r(b) - r(a) + w \geq 0$. If $r(a) < r(b) + w$, the dependency between a and b is respected. Otherwise $r(a) = r(b) + w$ and $<$ must respect $a < b$. As $a \overset{w}{\prec} b$, there is a path $c_0 = a, c_1, \ldots, c_{k-1}, c_k = b$ with total weigh w in G. For each $i \in [\![0,\, k-1]\!]$, $r(c_{i+1}) - r(c_i) + w_{i,i+1} < 0$ but:

$$\sum_{i=0}^{k-1} r(c_{i+1}) - r(c_i) + w_{i,i+1} = r(c_k) - r(c_0) + \sum_{i=0}^{k-1} w_{i,i+1} = r(b) - r(a) + w = 0$$

Thus $\forall i \in [\![0,\, k-1]\!]$, $r(c_{i+1}) - r(c_i) + w_{i,i+1} = 0$, thus the path from a to b is also present in G' thus $a < b$. □

A.3 Proof of Proposition 4

Proof. Let $\overset{n}{\precsim}$ be a valid weighted preorder and $\overset{n}{\simeq}$ the induce relation defined as in (4):

- $\overset{n}{\simeq}$ is reflexive. Indeed, $\forall a \in S :\ a \overset{0}{\precsim} a \wedge a \overset{-0}{\precsim} a$ thus $a \overset{0}{\simeq} a$.
- $\overset{n}{\simeq}$ is transitive. Let $x, y, z \in S$ such that $x \overset{n}{\simeq} y \wedge y \overset{m}{\simeq} z$. Then $x \overset{n+m}{\simeq} z$. Indeed:

$$\begin{aligned} x \overset{n}{\simeq} y \wedge y \overset{m}{\simeq} z \quad &\implies \quad x \overset{n}{\precsim} y \wedge y \overset{-n}{\precsim} x \wedge y \overset{m}{\precsim} z \wedge z \overset{-m}{\precsim} y \\ &\implies \quad x \overset{n+m}{\precsim} z \wedge z \overset{-n-m}{\precsim} x \\ &\implies \quad x \overset{n+m}{\simeq} z \end{aligned}$$

- Weights are unique. Let $x, y \in S, n, m \in \mathbb{Z}$. Lets assume $m > n$.

$$\begin{aligned} x \overset{n}{\simeq} y \wedge x \overset{m}{\simeq} y \quad &\implies \quad x \overset{n}{\precsim} y \wedge y \overset{-m}{\precsim} x \\ &\implies \quad x \overset{n-m}{\precsim} x \\ &\implies \quad \forall k \in \mathbb{N} :\ x \overset{k(n-m)}{\precsim} x \\ &\implies \quad \forall k \in \mathbb{Z} :\ x \overset{k}{\precsim} x \end{aligned}$$

  This violate the validity of $\overset{n}{\precsim}$. Thus $m \leq n$. By symmetry, $n \leq m$. Thus $n = m$.
- Weights are inversed with the operands:

$$\forall a, b \in S :\ a \overset{n}{\simeq} b \implies a \overset{n}{\precsim} b \wedge b \overset{-n}{\precsim} a \implies b \overset{-n}{\simeq} a$$

□

A.4 Proof of Remark 1

Proof. Let $\simeq$ be an weighted equivalence relation, X an equivalence class of $\simeq$, $a \in X$ and r a retiming defined over X as in the setup of Remark 1. Then:

$$\forall b \in X :\ b \overset{n}{\simeq} a \implies r(b) = n + r(a)$$

Let b, c in X such that $c \overset{n}{\simeq} a$ and $c \overset{m}{\simeq} b$. Then by definition of a weighted equivalence, $b \overset{n-m}{\simeq} a$ so $r(b) = n - m + r(a)$. Thus:

$$\begin{aligned} r(c) &= n + r(a) \\ &= n + r(a) + r(b) - r(b) \\ &= n + r(a) - (n - m + r(a)) + r(b) \\ &= m + r(b) \end{aligned}$$

Thus defining r from a is equivalent to defining it from b. □

A.5 Proof of Proposition 5

Let N be a node, I its inputs, O its outputs, V its variables and $\prec$ its dependency relation. Let $\precsim$ be a valid preorder respecting the dependencies of N. For every calling context C, let $G(C)$ be the node C where the call to N has be replaced by the grey-boxing induced by $\precsim$. The two directions of the equivalence of Proposition 5 are proved separately.

Static partitionings do not reject calling contexts

First we prove that if $\precsim$ is a static partitionings, then no valid calling context is rejected:

$$\left(\forall i \in I,\, o \in O :\; i \overset{n}{\preccurlyeq} o \iff i \overset{n}{\precsim} o\right) \implies (\forall C :\; causal(C) \implies causal(G(C)))$$

Proof. Lets assume $\forall i \in I,\, o \in O :\; i \overset{n}{\preccurlyeq} o \iff i \overset{n}{\precsim} o$. Let X be the sub-nodes induced by $\precsim$ and $depend_X$ the dependency relation between them. Let C be a node calling N, V_C its variables and $\prec_C$ its dependency relation. Let V_G and $\prec_G$ be the variables and the dependency relation of $G(C)$. Lets assume $\neg causal(G(C))$ and show $\neg causal(C)$.

As $\neg causal(G(C))$, exists $a \in V_G$ and $w \leq 0$ such that $a \overset{w}{\prec_G} a$. Then, by looking at the origin of the dependency cycle, we can find $a_0, \ldots, a_k \in V_G$ and $w_0, \ldots, w_{k-1} \in \mathbb{Z}$ such that, $a_0 = a_k$, $\exists i \in [\![0,\, k-1]\!] :$ $a_i = a$, $\sum_{i=0}^{k-1} w_i = w$ and $\forall i \in [\![0,\, k-1]\!]$:

- If $a_i, a_{i+1} \in V_C$, then $a_i \overset{w_i}{\prec_C} a_{i+1}$.
- If $a_i, a_{i+1} \in X$, then $a_i \overset{w_i}{\prec_X} a_{i+1}$.
- If $a_i \in V_C$ and $a_{i+1} \in X$, then $\exists i \in I$ such that i is mapped to a_i by the call to N in C and the retiming of i in the sub-node a_{i+1} is w_i.
- If $a_i \in X$ and $a_{i+1} \in V_C$, then $\exists o \in O$ such that o is mapped to a_{i+1} by the call to N in C and the retiming of o in the sub-node a_i is $-w_i$.

The $a_0, \ldots, a_{k-a}$ form a cycle of elements. In the following, we consider their indices to be taken modulo k to simplify notation. At least one of the elements of the cycle is in V_C. Otherwise we would have dependency cycle in $\prec_X$ but this is impossible as explained in Section 4.5.2. Thus every contiguous group of sub-nodes in the circle is surrounded by elements of C. Thus they can be replaced by a dependency already existing in C. Indeed, for $j_0, j_1 \in [\![0,\, k-1]\!]$ such that:

$$a_{j_0}, a_{j_1} \in V_C \wedge \forall j \in [\![j_0 + 1,\, j_1 - 1]\!] :\; a_j \in X$$

there exist $i \in I, o \in O$ such that i is mapped to a_{j_0} and o to a_{j_1} in the call to N. Moreover, i and o are respectively in the sub-nodes a_{j_0+1} and a_{j_1-1} of N with a retiming of w_{j_0} and $-w_{j_1-1}$. Let $w_{io} = \sum_{j=j_0}^{j_1-1} w_j$. Then, if w_{io} using the transitivity of $\precsim$ and the definition of $\preccurlyeq_X$, we can deduce $i \overset{w_{io}}{\precsim} o$. Using the initial assumption that $\precsim$ is a static partitioning, this gives us $i \overset{w_{io}}{\preccurlyeq} o$. Thus $a_{j_0} \overset{w_{io}}{\prec_C} a_{j_1}$.

This allows us to find a dependency cycle in C with the same weight. Thus we have $\neg causal(C)$ which was our proof goal. □

Static schedules that do not reject any calling context are static partitionings

Then we prove that if $\precsim$ does not reject any valid calling context, then it is a static partitioning:

$$(\forall C :\; causal(C) \implies causal(G(C))) \implies \left(\forall i \in I,\, o \in O :\; i \overset{n}{\preccurlyeq} o \iff i \overset{n}{\precsim} o\right)$$

Proof. Lets assume:

$$\neg\left(\forall i \in I,\, o \in O :\; i \overset{n}{\preccurlyeq} o \iff i \overset{n}{\precsim} o\right) \tag{12}$$

and show $\neg\left(\forall C :\; causal(C) \implies causal(G(C))\right)$ or equivalently $\exists C :\; causal(C) \wedge \neg causal(G(C))$. As $\precsim$ respect dependencies, (12) gives us $i \in I$, $o \in O$ and $n \in \mathbb{Z}$ such that:

$$i \overset{n}{\precsim} o \wedge \neg\left(i \overset{n}{\preccurlyeq} o\right)$$

Then the context that just replicates the inputs and the outputs of N to its inputs and outputs except for i that is defined by $i(t) = o(t) - n$ creates a dependency cycle in $G(C)$ but not in C. Indeed, the dependency graph of C only contains the dependencies between the inputs and outputs of N and the edge $o \xrightarrow{n} i$. But as $\neg i \overset{n}{\preccurlyeq} o$ this does not create a dependency cycle. On the opposite, $i \overset{n}{\precsim} o$ and thus $G(C)$ contains the dependency $i \overset{n}{\preccurlyeq} o$. □

A.6 Proof of Proposition 6

Proof. Let $x, y \in S$ such that $x \overset{n}{\simeq} y$. Thus $x \overset{n}{\precsim} y \wedge y \overset{-n}{\precsim} x$. Let $i \in I, o \in O$.

$$\begin{aligned}
i \overset{m}{\preccurlyeq} x \wedge y \overset{k}{\preccurlyeq} o \quad &\implies \quad i \overset{m}{\precsim} x \wedge y \overset{k}{\precsim} o && (13)\\
&\implies \quad i \overset{m}{\precsim} x \overset{n}{\precsim} y \overset{k}{\precsim} o && (14)\\
&\implies \quad i \overset{m+n+k}{\precsim} o && (15)\\
&\implies \quad i \overset{m+n+k}{\preccurlyeq} o && (16)
\end{aligned}$$

The first implication is obtained from (5) and the last from (6). Similarly, we have $i \overset{m}{\preccurlyeq} y \wedge x \overset{b}{\preccurlyeq} o \implies i \overset{m-n+k}{\preccurlyeq} o$. Thus $x \overset{n}{\chi} y$. □

A.7 Proof of Proposition 7

Proof. We first check $\overset{n}{\precsim}_{\mathcal{I}}$ is a static partitioning.

- It is transitive. Indeed, let $x, y, z \in Var$.

$$\begin{aligned}
x \overset{n}{\precsim}_{\mathcal{I}} y \wedge y \overset{m}{\precsim}_{\mathcal{I}} z \quad &\implies \quad \mathcal{I}_0^{\precsim}(x) \subseteq \mathcal{I}_n^{\precsim}(y) \wedge \mathcal{I}_0^{\precsim}(y) \subseteq \mathcal{I}_m^{\precsim}(z)\\
&\implies \quad \mathcal{I}_0^{\precsim}(x) \subseteq \mathcal{I}_n^{\precsim}(y) \wedge \mathcal{I}_n^{\precsim}(y) \subseteq \mathcal{I}_{m+n}^{\precsim}(z)\\
&\implies \quad \mathcal{I}_0^{\precsim}(x) \subseteq \mathcal{I}_{m+n}^{\precsim}(z)\\
&\implies \quad x \overset{n+m}{\precsim} z
\end{aligned}$$

- It contains all dependency constraints. Indeed, let $x, y \in Var$.

$$\begin{aligned} x \overset{n}{\prec} y &\implies x \overset{n}{\precsim} y \\ &\implies \forall i \in I, m \in \mathbb{Z} : i \overset{m}{\precsim} x \implies \left(i \overset{m}{\precsim} x \wedge x \overset{n}{\precsim} y \right) \\ &\implies \forall i \in I, m \in \mathbb{Z} : i \overset{m}{\precsim} x \implies i \overset{n+m}{\precsim} y \\ &\implies \mathcal{I}_0^{\precsim}(x) \subseteq \mathcal{I}_n^{\precsim}(y) \\ &\implies x \overset{n}{\precsim_{\mathcal{I}}} y \end{aligned}$$

  In particular, it is reflexive for any positive weight as $\overset{n}{\prec}$ respect this property.

- It is valid. Indeed, let $x, y \in Var$. Since every equation depend on an input (see Section 4.6), $\exists i_0 \in I, m \in \mathbb{Z} : i_0 \overset{m}{\precsim} x$

$$\begin{aligned} x \overset{n}{\precsim_{\mathcal{I}}} y &\implies \forall i \in I, m \in \mathbb{Z} : i \overset{m}{\precsim} x \implies i \overset{m+n}{\precsim} y \\ &\implies i_0 \overset{m}{\precsim} x \implies i_0 \overset{n+m}{\precsim} y \\ &\implies i_0 \overset{n+m}{\precsim} y \end{aligned}$$

  As $\overset{n}{\precsim}$ is valid, $\exists n_0 : x \overset{n}{\precsim_{\mathcal{I}}} y \implies n \geq n_0$.

- It strictly maps dependencies on input/output pairs. Indeed, let $i \in I, o \in O$.

$$\begin{aligned} i \overset{n}{\precsim_{\mathcal{I}}} o &\implies \forall i' \in I, m \in \mathbb{Z} : i' \overset{m}{\precsim} i \implies i' \overset{n+m}{\precsim} o \\ &\implies i \overset{0}{\precsim} i \implies i \overset{n}{\precsim} o \\ &\implies i \overset{n}{\precsim} o \\ &\implies i \overset{n}{\prec} o \end{aligned}$$

Then we check the optimality of $\overset{n}{\precsim_{\mathcal{I}}}$ on output pairs. Let $o_1, o_2 \in O$.

$$\begin{aligned} o_1 \overset{n}{\chi} o_2 &\implies \forall i \in I, o \in O : \left(i \overset{m}{\preccurlyeq} o_1 \wedge o_2 \overset{k}{\preccurlyeq} o \implies i \overset{m+n+k}{\preccurlyeq} o \right) \wedge \left(i \overset{m}{\preccurlyeq} o_2 \wedge o_1 \overset{k}{\preccurlyeq} o \implies i \overset{m-n+k}{\preccurlyeq} o \right) \\ &\implies \forall i \in I : \left(i \overset{m}{\preccurlyeq} o_1 \implies i \overset{m+n}{\preccurlyeq} o_2 \right) \wedge \left(i \overset{m}{\preccurlyeq} o_2 \implies i \overset{m-n}{\preccurlyeq} o_1 \right) \\ &\implies \mathcal{I}_0^{\precsim}(o_1) \subseteq \mathcal{I}_n^{\precsim}(o_2) \wedge \mathcal{I}_0^{\precsim}(o_2) \subseteq \mathcal{I}_{-n}^{\precsim}(o_1) \\ &\implies o_1 \overset{n}{\precsim_{\mathcal{I}}} o_2 \wedge o_2 \overset{-n}{\precsim_{\mathcal{I}}} o_1 \\ &\implies o_1 \overset{n}{\simeq_{\mathcal{I}}} o_2 \end{aligned}$$

The validity and optimality of $\overset{n}{\precsim_{\mathcal{O}}}$ can be proved in a similar manner. □

A.8 Proof of Proposition 8

Proof. Let $i \in I$, $o \in O$ and $n \in \mathbb{Z}$.

$$
\begin{aligned}
i \overset{n}{\chi} o \quad &\Longrightarrow \quad \forall i' \in I, o' \in O : \left(i' \overset{m}{\preccurlyeq} i \wedge o \overset{k}{\preccurlyeq} o' \Rightarrow i' \overset{m+n+k}{\preccurlyeq} o' \right) \wedge \left(i' \overset{m}{\preccurlyeq} o \wedge i \overset{k}{\preccurlyeq} o' \Rightarrow i' \overset{m-n+k}{\preccurlyeq} o' \right) \\
&\Longrightarrow \quad \left(i \overset{0}{\preccurlyeq} i \wedge o \overset{0}{\preccurlyeq} o \Rightarrow i \overset{n}{\preccurlyeq} o \right) \wedge \forall i' \in I, o' \in O : i' \overset{m}{\preccurlyeq} o \Rightarrow \left(i \overset{k}{\preccurlyeq} o' \Rightarrow i' \overset{m-n+k}{\preccurlyeq} o' \right) \\
&\Longrightarrow \quad i \overset{n}{\preccurlyeq} o \wedge \forall i' \in I, o' \in O : i' \overset{m}{\precsim} o \Rightarrow \left(i \overset{k}{\precsim} o' \Rightarrow i' \overset{m-n+k}{\precsim} o' \right) \\
&\Longrightarrow \quad i \overset{n}{\precsim} o \wedge \forall i' \in I \left(o \overset{0}{\precsim} o \Rightarrow i' \overset{m}{\precsim} o \right) \Rightarrow \left(\mathcal{O}_0^{\precsim}(i) \subseteq \mathcal{O}_{m-n}^{\precsim}(i') \right) \\
&\Longrightarrow \quad i \overset{n}{\precsim} o \wedge \forall i' \in I : \left(\mathcal{O}_0^{\precsim}(o) \subseteq \mathcal{O}_m^{\precsim}(i') \right) \Rightarrow \left(\mathcal{O}_0^{\precsim}(i) \subseteq \mathcal{O}_{m-n}^{\precsim}(i') \right) \\
&\Longrightarrow \quad i \overset{n}{\precsim} o \wedge \forall i' \in I : i' \overset{m}{\precsim}_{\mathcal{O}} o \Rightarrow' \overset{m-n}{\precsim}_{\mathcal{O}} i \\
&\Longrightarrow \quad i \overset{n}{\precsim} o \wedge \mathcal{I}_0^{\precsim_{\mathcal{O}}}(o) \subseteq \mathcal{I}_{-n}^{\precsim_{\mathcal{O}}}(i) \\
&\Longrightarrow \quad i \overset{n}{\precsim}_{\mathcal{I}_{\mathcal{O}}} o \wedge o \overset{-n}{\precsim}_{\mathcal{I}_{\mathcal{O}}} i \\
&\Longrightarrow \quad i \overset{n}{\simeq}_{\mathcal{I}_{\mathcal{O}}} o
\end{aligned}
$$

□