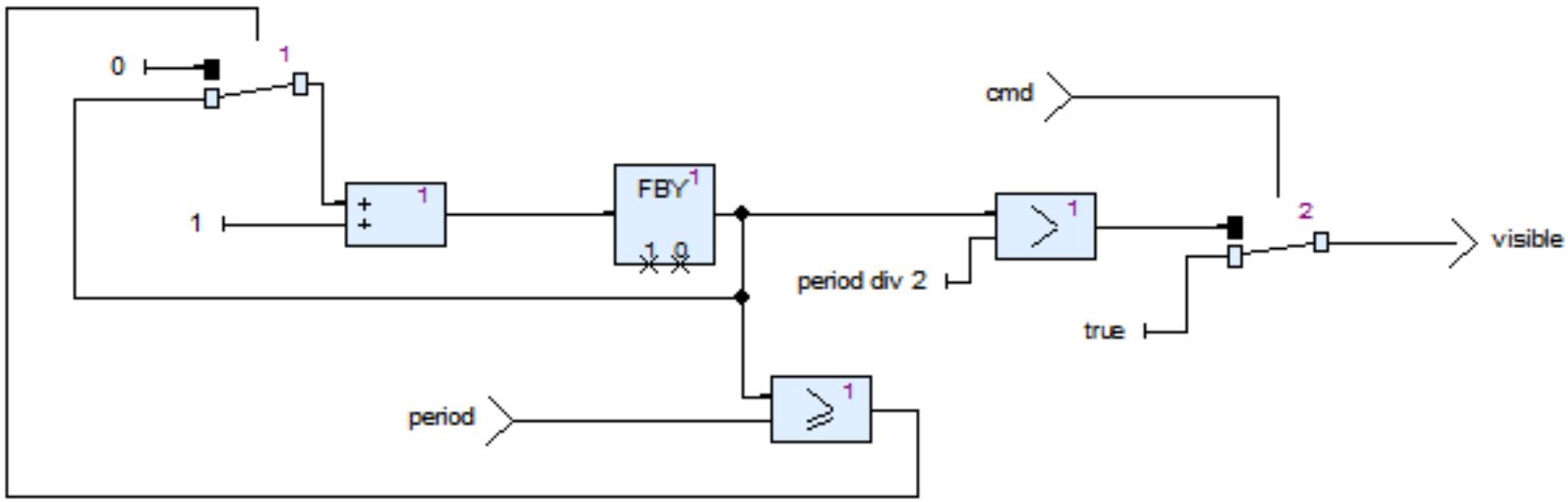
In-Place Array Update in a Dataflow Synchronous Language

Ulysse Beaugnon, Albert Cohen, Marc Pouzet

École Normale Supérieure, INRIA

Synchronous Block Diagrams - SCADE[1]



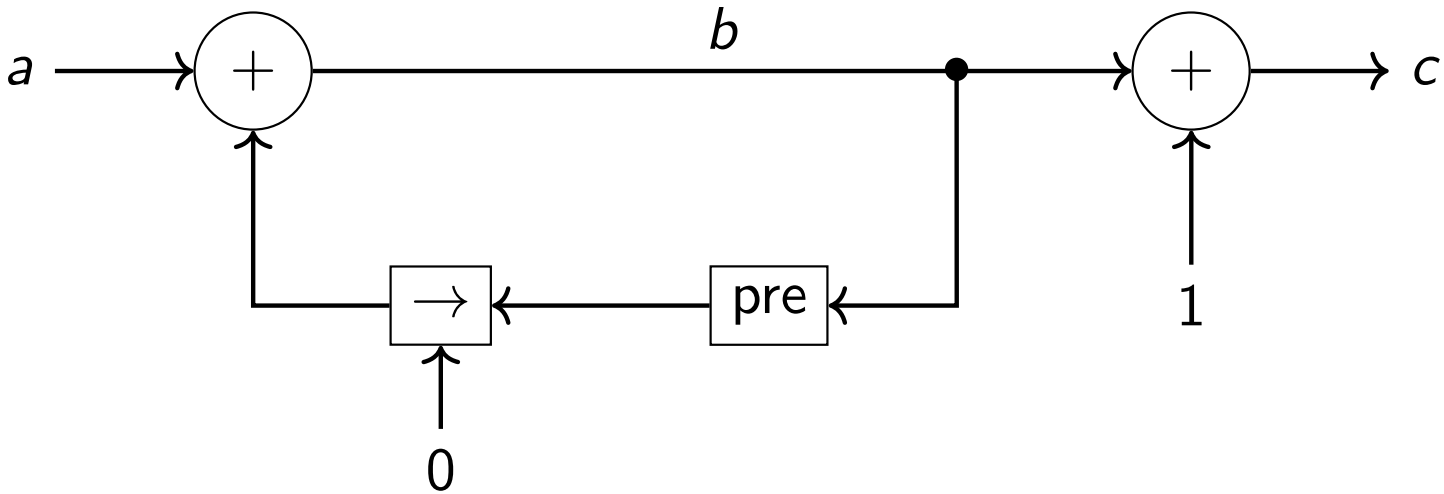
- DSL for modeling/implementing real-time control software
- Wires define streams of values
- All nodes progress at the same speed

[1] http://www.esterel-technologies.com/

Dataflow Synchronous Languages - Lustre

```
node foo(a: int) = (c: int) {
  c = b + 1                    (* c_t = b_t + 1 *)

  b = a + (0 -> pre b)         (* b_0 = a_0 *)
                               (* b_t = a_t + b_{t-1} *)
}
```



Equation Ordering

```
node foo(a: int) = (c: int) {
  c = b + 1                    (* c_t = b_t + 1 *)

  b = a + (0 -> pre b)         (* b_0 = a_0 *)
                               (* b_t = a_t + b_{t-1} *)
}
```

Implicit ordering of equations given by data dependencies

- Ordering in source code doesn't matter

Allow feeback loops

- Every dependency loop must cross a delay

Compilation to efficient C code

```
node foo(a: int) = (c: int) {
  c = b + 1                    (* c_t = b_t + 1 *)

  b = a + (0 -> pre b)         (* b_0 = a_0 *)
                               (* b_t = a_t + b_{t-1} *)
}
```

Generated C code

```
// Computes a time step of foo
void foo_step(foo_state_t* state, int a, int* c) {
  int b = a + ((t == 0) ? 0 : state->b);
  *c = b + 1;

  state->b = b; // Update state
}
```

Functional Arrays

- Declare a new array and read from it

```
e = d^1000
f = e[4]
```

- Define a new array from an old one

```
g = e[3] <- 42
```

Functional Arrays

- Declare a new array and read from it

```
e = d^1000
f = e[4]
```

- Define a new array from an old one

```
g = e[3] <- 42
```

Immutable arrays: no in-place update

```
// Generated C code
int e[1000] = { d };
int f = e[4];

int g[1000];
memcpy(g, e, 1000*sizeof(int)); <-- Costly operation
g[3] = 42;
```

Problem

How to avoid array copies ...

... while keeping functional semantics ?

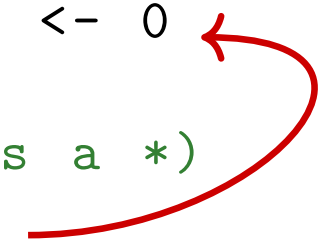
Destructive Updates

Ensure no array is accessed after being updated

- Ensure arrays are updated only once
- Add dependencies from reads to writes
- Let the scheduling algorithm do the job

```
(* Consume a, reuse its memory for b *)
b = a[0] <- 0

(* Access a *)
c = a[0]
```



c must be executed before *b*

Copies are no longer needed

- Modifications of the original array cannot be observed
- Reuse the memory of the original array

Strenghs of the Destructive Update Approach

Keeps pure functional semantics

Removes all implicit copies

- Reject programs that cannot be implemented without copies
- Explicit copies with the `copy` operator if needed
- No hidden performance cost

Direct mapping from source to generated code

- Only dependency analysis is more complex
- Required by certification authorities

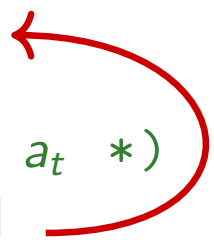
Inter-Reaction Copies

What about `pre` ? Two solutions:

1. Insert a copy at every `pre`
2. **Handle inter-reaction aliasing**

```
(* b_t consumes a_t *)
b = a[0] <- 0

(* c_{t+1} accesses a_t *)
c = (pre a)[0]
```



c_{t+1} must be executed before b_t

Must *retime* equations

- Compute c_t at time $t - 1$

Problems to Solve

Inter-Reaction Alias Analysis

- Avoid unnecessary explicit copies

Array Memory Management

- Arrays outlive a single time step

Modular Compilation

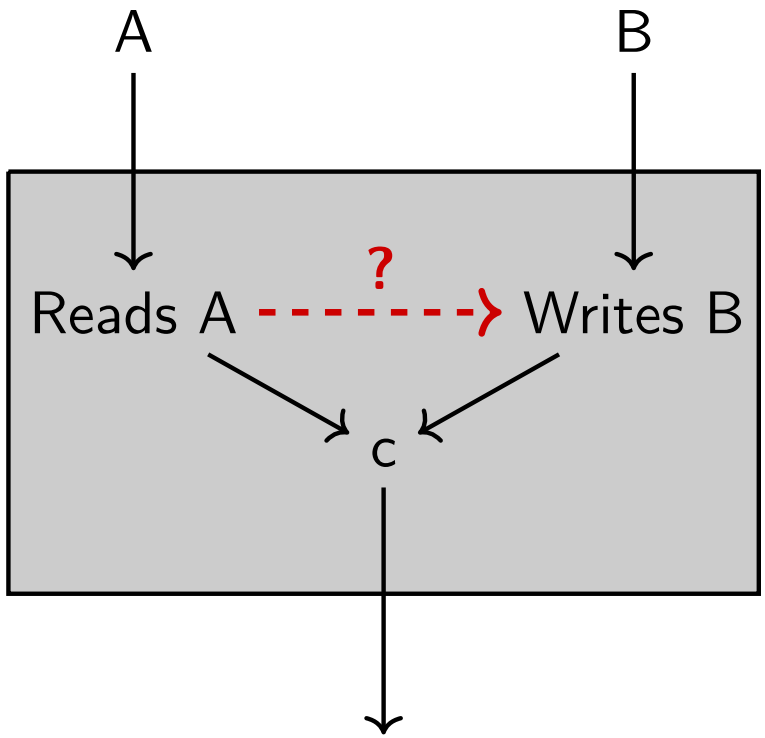
- Compile a node independently of its calling context
- Unknown aliasing between arguments
- Retiming imposed by feedback loops
- The alternative is inlining: exponential code size

Modular Compilation - Unknown Aliasing

```
node f(A, B: int[8])
     = (c: int) {
   D = B[0] <- 0
   c = A[3] + D[3]
}

(* Without aliasing *)
x = f(A', B')

(* With aliasing *)
y = f(A', A')
```



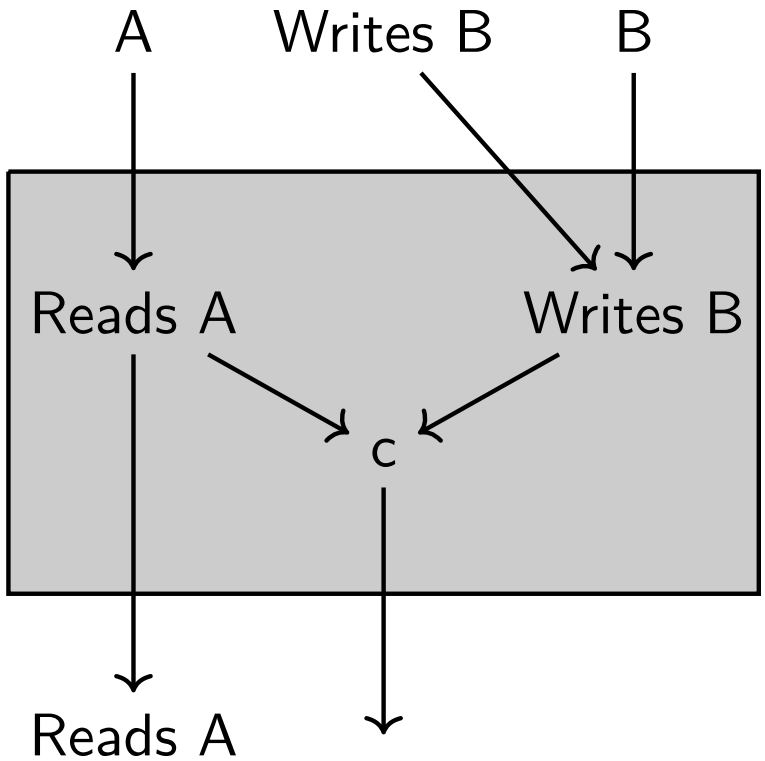
Is there a dependency from `Read A` to `Write B` ?

Modular Compilation - Unknown Aliasing

```
node f(A, B: int[8])
     = (c: int) {
  D = B[0] <- 0
  c = A[3] + D[3]
}

(* Without aliasing *)
x = f(A', B')

(* With aliasing *)
y = f(A', A')
```



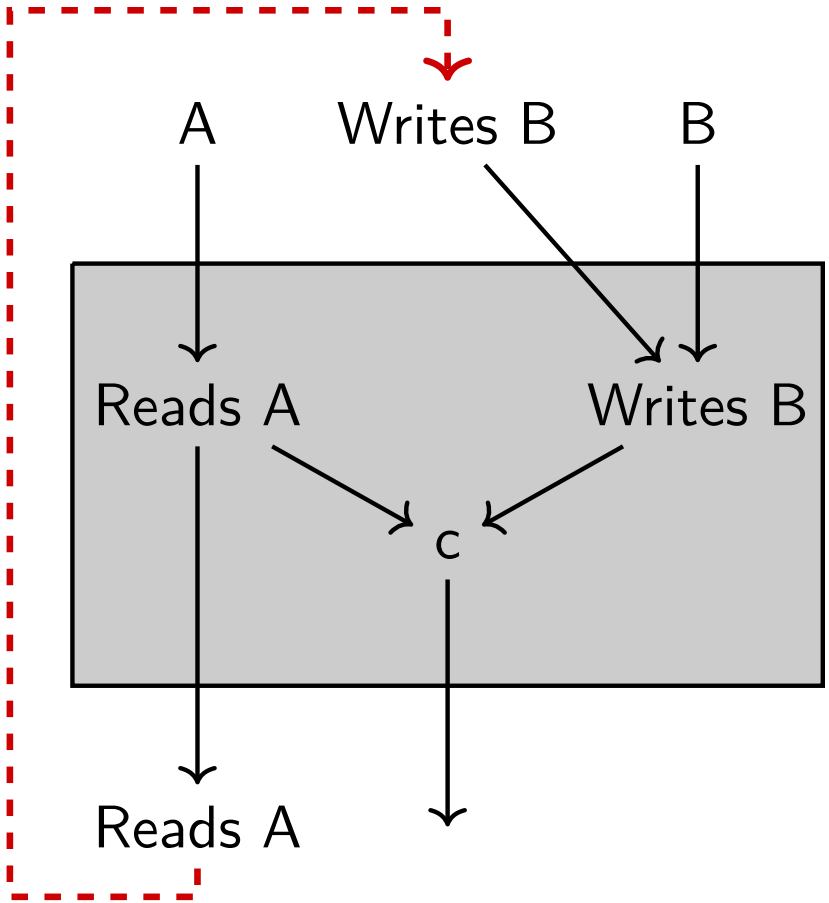
Expose reads and writes to the calling context

Modular Compilation - Unknown Aliasing

```
node f(A, B: int[8])
     = (c: int) {
   D = B[0] <- 0
   c = A[3] + D[3]
}

(* Without aliasing *)
x = f(A', B')

(* With aliasing *)
y = f(A', A')
```



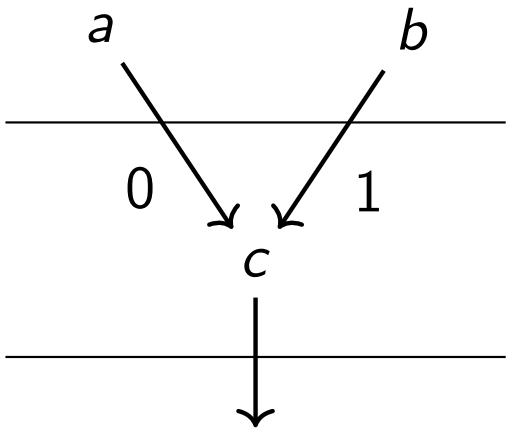
The context adds feedback loops if needed

Dependency Graph

Dependencies are of the form:

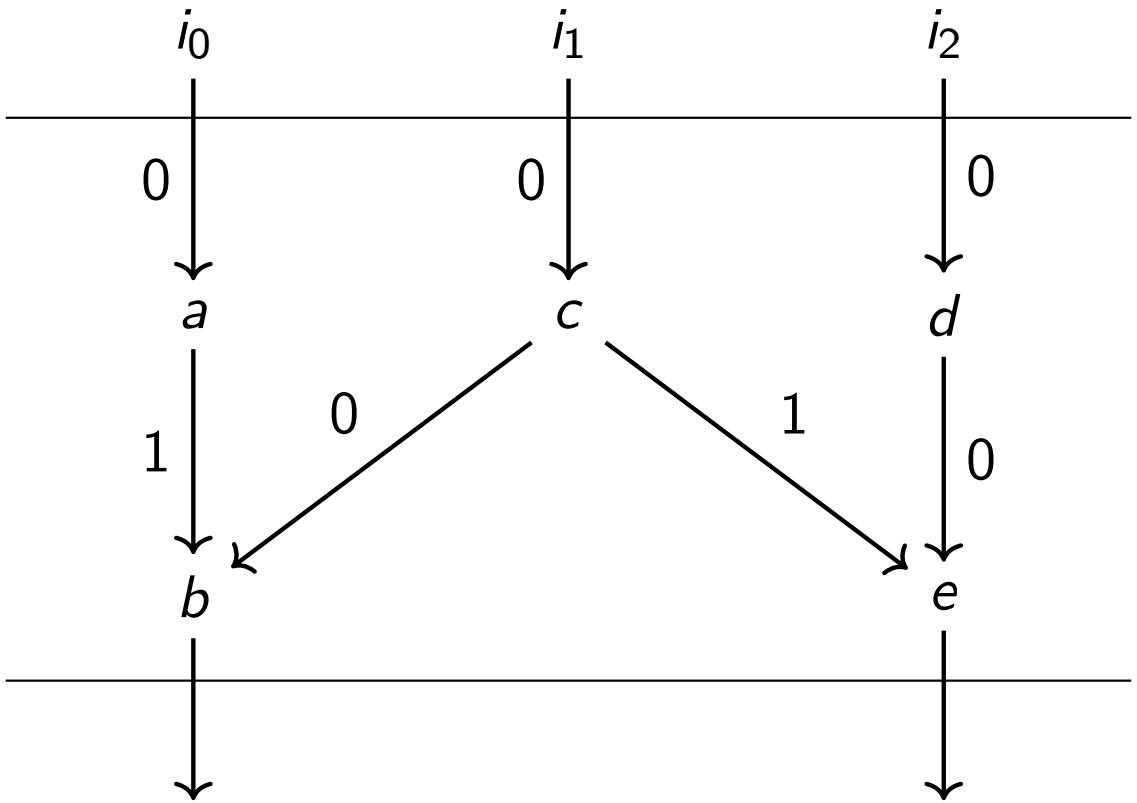
$$\forall t \in \mathbb{N} : \; a_t \text{ depends on } b_{t-w} \qquad \text{with } w \in \mathbb{Z} \text{ constant.}$$

Dependencies represented by a weighted graph:



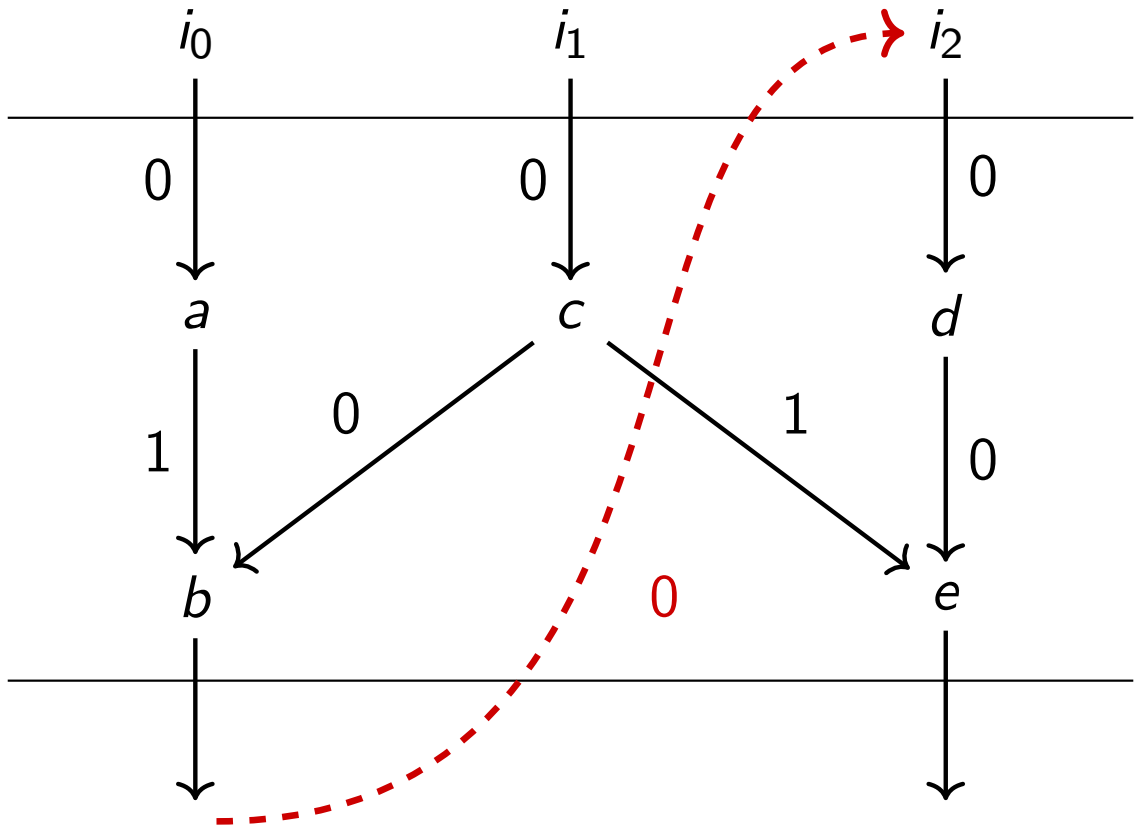

```
node foo(a, b: int) = (c: int) {
  (* c_t = a_t + b_{t-1} *)
  c = 0 -> a + pre b
}
```

Modular Compilation - Feedback Loops



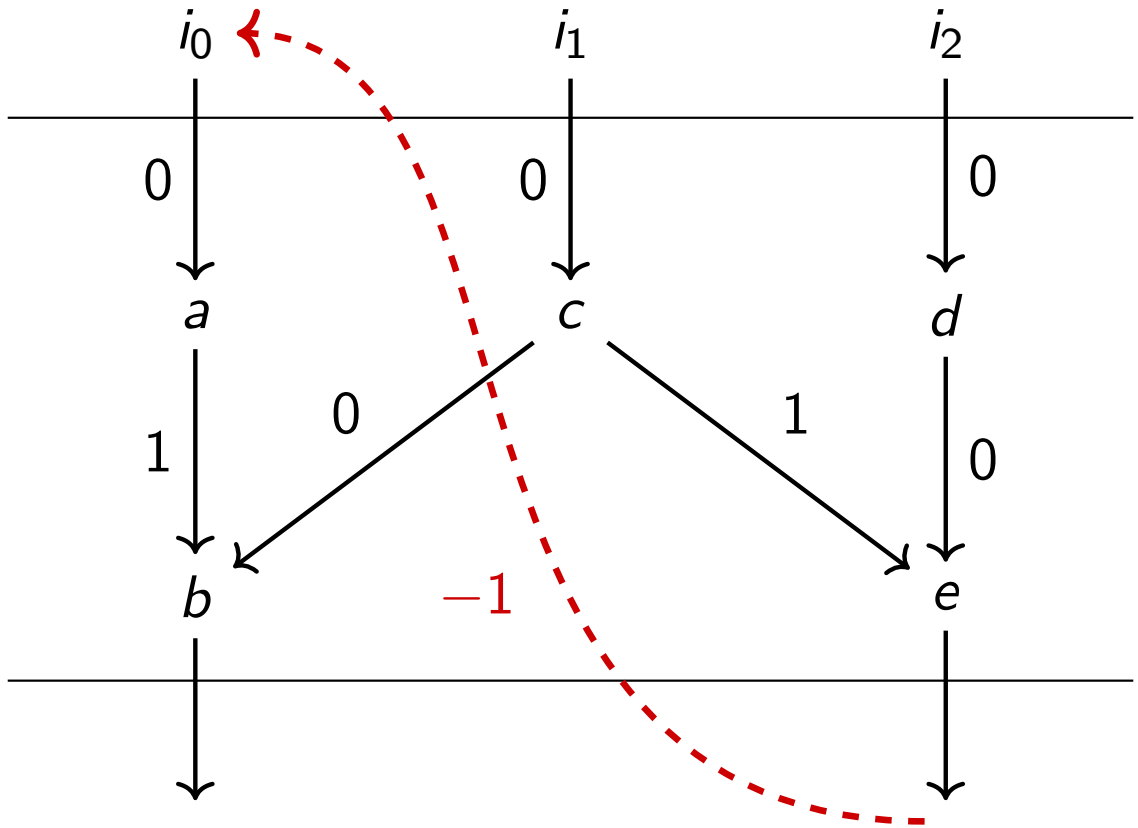
How can we schedule this node ?

Modular Compilation - Feedback Loops



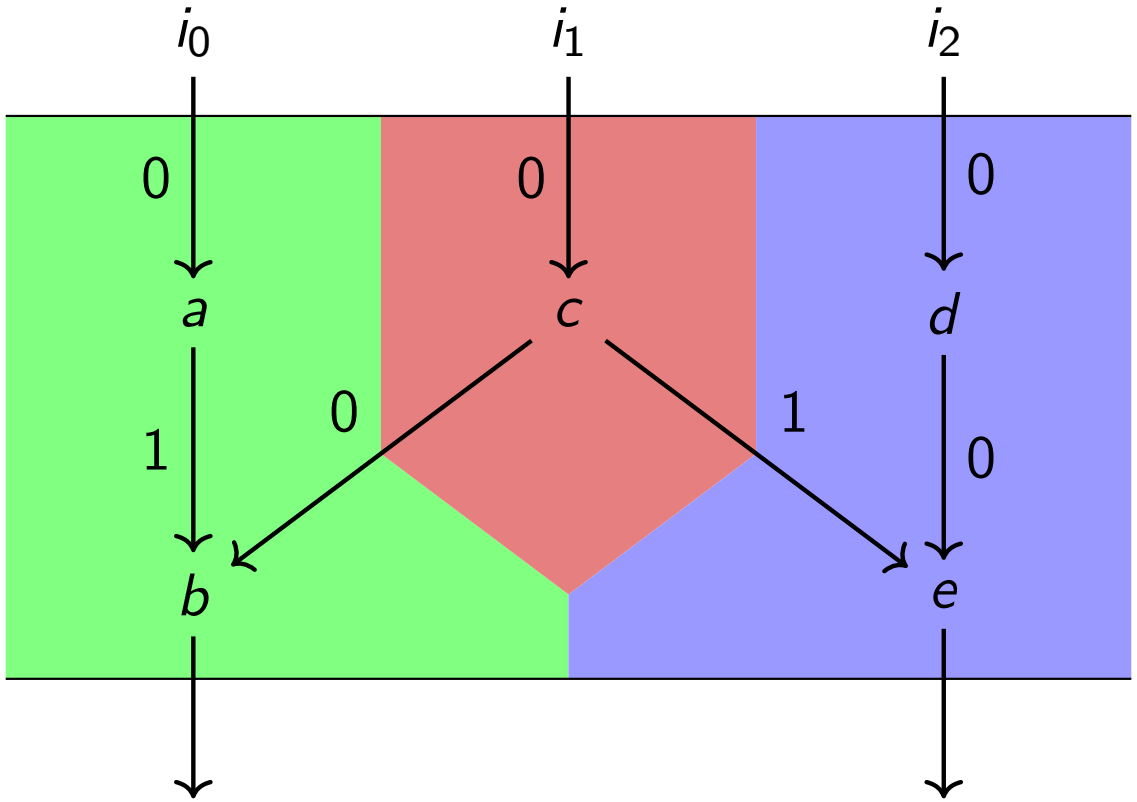
$\{a_{t-1}, b_t\}$ before $\{d_t, e_t\}$

Modular Compilation - Feedback Loops



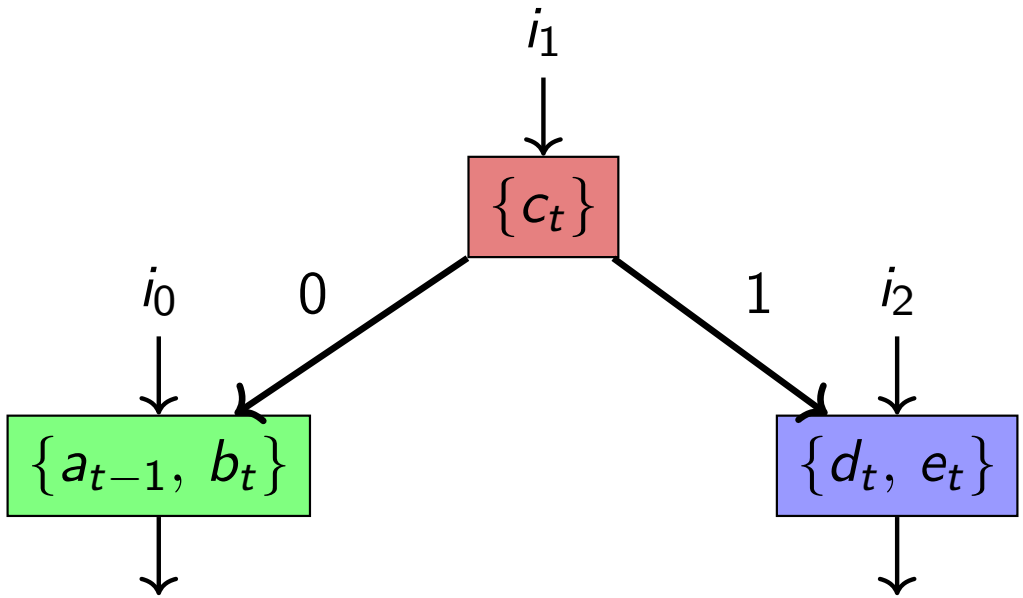
$\{d_t,\ e_t\}$ before $\{a_{t-1},\ b_t\}$

Modular Compilation - Feedback Loops



- a_{t-1} and b_t can be executed atomically
- c_t can be executed atomically
- d_t and e_t can be executed atomically

Grayboxing



Grayboxing: Partioning into atomic subnodes

- Subnodes are compiled independently
- The calling context orders subnodes
- Avoid a full inlining

Grayboxing Definition

A grayboxing is given by:

- A partitioning $X^0, \ldots, X^{k-1}$ of equations in atomic sub-nodes
- A retiming function for each sub-node $r_i : X_i \to \mathbb{Z}$
  - a_t is computed at the reaction $t + r(a)$
- A dependency relation $X \xrightarrow{w} Y$ on sub-nodes
  - $X \xrightarrow{w} Y \implies Y_t$ depends on X_{t-w}

A Grayboxing must:

- Respect dependencies between equations
- Not reject any calling context

Extension of *[Pouzet and Raymond 2009]* for retiming

Finding a Minimal Grayboxing

Goal: Minimize the Number of Partitions

Optimal Solution: NP-Complete

- Encode the problem for an SMT solver

Heuristic: Find a Good-Enough Partitioning

- Optimal on inputs and outputs
- Based on an existing heuristic that doesn't handle retiming

See the paper for more information

Conclusion

In-place updates in a synchronous dataflow language

- Avoid copy operations
- Keeps pure functional semantics
- No hidden performance cost
- No expressivity loss

Relies on scheduling constraints

- Ensure no array is accessed after being written to
- Leverages the exsiting scheduling algorithms

Destructive Updates for Synchronous Dataflow Languages

With copying `pre`

- Minimal alteration of the compilation process
- Arrays copied at the end of iterations

With inter-iteration aliasing

- `pre` creates aliasing instead of copying
- Need for retiming created by iter-iteration aliasing
- Context-aware scheduling and retiming for more genericity
- Modular compilation enabled by the grayboxing technique