Destructive updates for a synchronous language

Ulysse Beaugnon

Under the supervision of Albert Cohen and Marc Pouzet

December 2, 2014

Efficient arrays for a synchronous dataflow language

- Based on a Lustre-like synchronous dataflow language
- Improve array performance
- Keep a functional semantics
- Preserve the modularity of functions

Functional arrays

Synchronous dataflow code

```
(* Declares a new array and reads from it *)
e: int[10] = d^10
f: int = d[4]

(* Defines a new array from e *)
g: int[10] = e[3] <- 42
```

C code

```
int e[10] = { d }, g[10];
int f = e[4];

memcpy(g, e, 10*sizeof(int));  ⟵ costly
g[3] = 42;
```

Outline

Destructive updates

Modular scheduling

Scheduling algorithms

Conclusion and future work

Destructive updates

Performance issues with functional arrays

- Each update implies a full copy of the array
- The copy is needed only if the original array is accessed later

```
(* g is a new array *)
g: int[10] = e[3] <- 42
```

Destructive updates with a functional semantics

- When an array is written to, it is consumed
- Add constraints to ensure no consumed array is accessed
- Avoid copies and keep a functional semantic

Constraints as dependencies

Our approach: scheduling constraints

- Add dependencies from reads to aliasing writes
- Rely on the implicitly scheduled semantic of the language

```
(* Consumes a *)
b: int[8] = a[0] <- 0

(* Accesses a *)
c: int = a[0] + 10
```

b depends on *c*

Unavoidable copies

- Reject programs when dependency cycles are detected
- Let the programmer manually add copies

Retiming

```
(* Original code: b(t) depends on C(t+1) *)
B: int[8] = (pre A)[i] <- 3
c: int    = A[4] + 10
```

```
(* Generated code: c is delayed by 1 *)
pre_c: int = (pre A)[4] + 10
B: int[8]  = (pre A)[i] <- 3
```

Retiming is needed for genericity

- Retiming depends on aliasing
- Aliasing depends on the calling context
- Generate different retimings for different contexts

Static schedule

A static schedule is given by:

- A retiming function $r : Eq \to \mathbb{Z}$
  - $a(t)$ is computed at the reaction $t + r(a)$
- A total order $\lhd$ on Eq to schedule within a single reaction
  - $a \lhd b \iff a$ is scheduled before b

Data and R/W dependencies must be respected

$$\forall t \in \mathbb{N} : \; a(t) \text{ depends on } b(t - w) \implies$$
$$r(b) < r(a) + w \vee (r(b) = r(a) + w \wedge b \lhd a)$$

Outline

Destructive updates

Modular scheduling

Scheduling algorithms

Conclusion and future work

Separate compilation

Need the context to compile

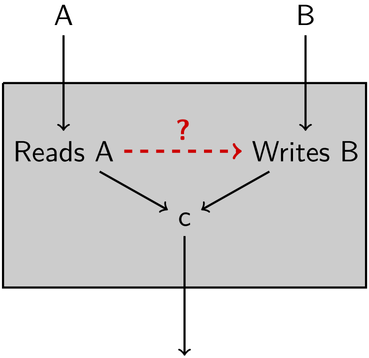
- Aliasing between arguments
- Feedback loops might add dependencies
- Need dependencies to order equations

Try to avoid inlining

- Exponential compilation time
- Exponential generated code size

Aliasing dependencies as feedback loops

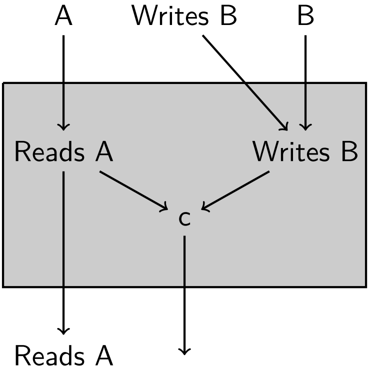
```
node f(A, B: int[8])
    = (c: int) {
  D: int[8] = B[0] <- 0
  c: int = A[3] + D[3]
}

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

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



Aliasing unknown

Aliasing dependencies as feedback loops

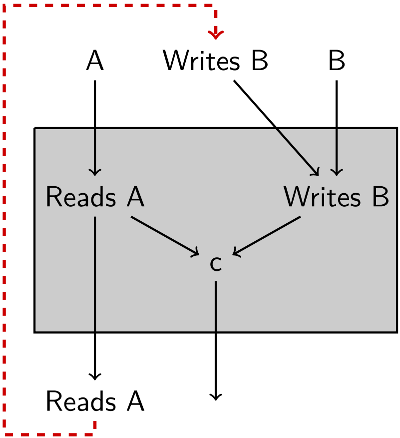
```
node f(A, B: int[8])
    = (c: int) {
  D: int[8] = B[0] <- 0
  c: int = A[3] + D[3]
}

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

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



Only expose arrays that alias with an input or an ouput

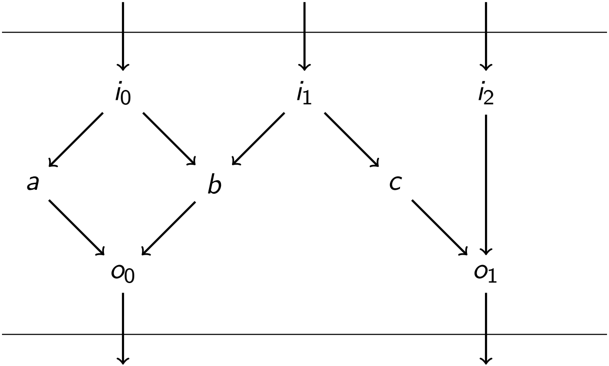
Aliasing dependencies as feedback loops

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

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

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



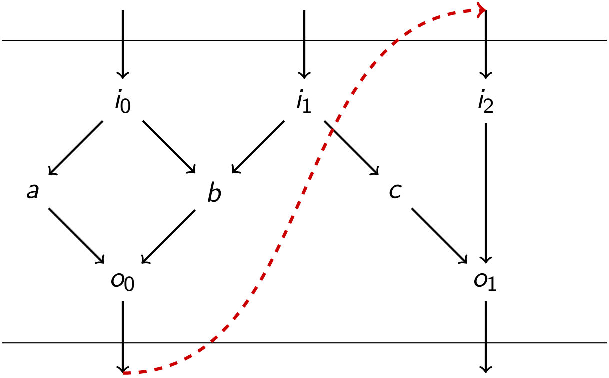
Reduced to the the problem of feeback loops handling

Feedback loops without retiming



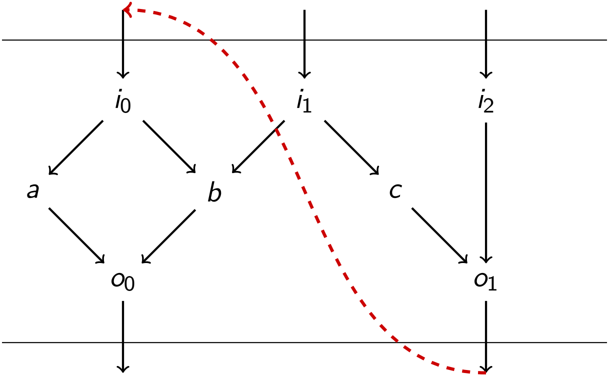
Dependency graph representing a synchronous dataflow function

Feedback loops without retiming



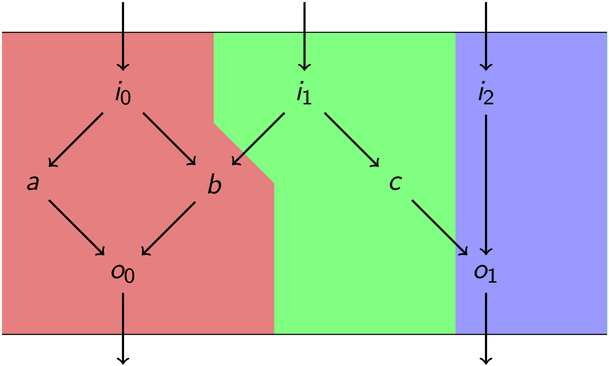
The schedule depends on the context

Feedback loops without retiming



The schedule depends on the context

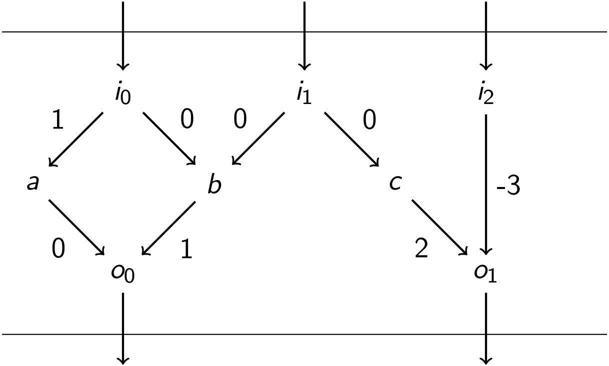
Feedback loops without retiming



P. Raymond & M. Pouzet: Grey-boxing

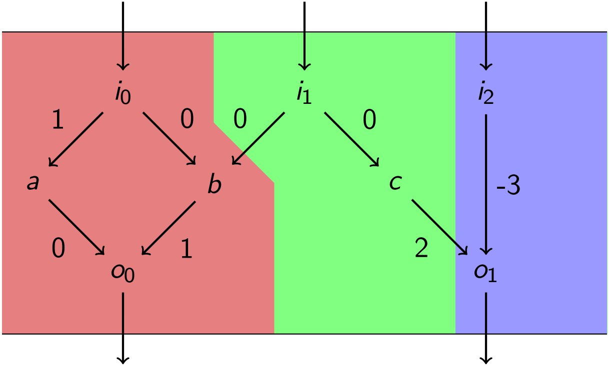
- compile atomic groups of equations together
- only keep dependencies between the groups

Feedback loops with retiming



$$b \xrightarrow{w} a \iff a(t) \text{ depends on } b(t-w)$$

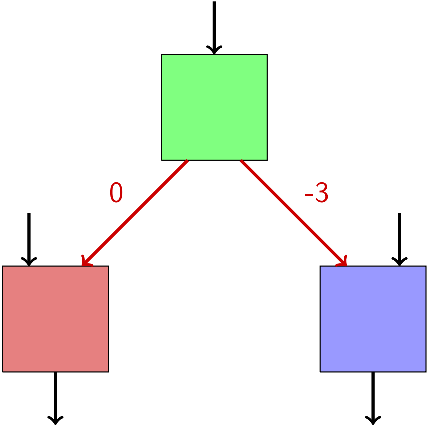
Feedback loops with retiming



$i_0(t),\ a(t-1),\ b(t),\ o_0(t-1)$

$i_1(t),\ c(t)$ $i_2(t),\ o_1(t+3)$

Feedback loops with retiming



Only keep the I/O, the dependencies and the retiming constraints

Grey-boxing formalization

A grey-boxing is given by:

- A partitioning $X_0, \ldots, X_{k-1}$ of equations in atomic sub-nodes
- A static schedule $(r_i : X_i \to \mathbb{Z}, \lhd_i)$ for each sub-node
- A dependency relation $X_i \xrightarrow{w} X_j$ among sub-nodes

Grey-boxings must:

- Not forbid any calling context
- Respect dependencies. If $a \in X_i$ and $b \in X_j$, then:

$$a(t) \text{ depends on } b(t-w) \implies X_j \xrightarrow{w - r_j(b) + r_i(a)} X_i$$

Outline

Destructive updates

Modular scheduling

Scheduling algorithms

Conclusion and future work

Encode a grey-boxing as a relation

Dependency relation: strict ordering

$$a \xrightarrow{w} b \iff a(t - w) \text{ must be scheduled before } b(t)$$

Weighted preorder: allows equations to be grouped

$$a \overset{w}{\precsim} b \iff a(t - w) \text{ is scheduled before or with } b(t)$$

Weighted equivalence: gives the groups and their retiming

$$\begin{aligned} a \overset{w}{\simeq} b \quad &\iff \quad a \overset{w}{\precsim} b \wedge b \overset{-w}{\precsim} a \\ &\iff \quad a \text{ and } b \text{ are in the same group and } r(a) - r(b) = w \\ &\iff \quad b(t) \text{ is computed together with } a(t - w) \end{aligned}$$

Weighted preorder

Definition (Weighted preorder)

A *weighted preorder* $\precsim$ is a ternary relation $\subseteq S \times \mathbb{Z} \times S$ that:

- is reflexive for any positive weight:

$$\forall a \in S, w \geq 0 : \ a \overset{w}{\precsim} a$$

- is transitive:

$$\forall a, b, c \in S : \ a \overset{v}{\precsim} b \wedge b \overset{w}{\precsim} c \implies a \overset{v+w}{\precsim} c$$

Weighted equivalence

Definition (Weighted equivalence)

A *weighted equivalence* $\simeq$ is a ternary relation $\subseteq S \times \mathbb{Z} \times S$ that:

- is relfexive: $\forall a \in S : \; a \overset{0}{\simeq} a$
- is transitive:
$$\forall a, b, c \in S : \; a \overset{v}{\simeq} b \wedge b \overset{w}{\simeq} c \implies a \overset{v+w}{\simeq} b$$
- has unique weights:
$$\forall a, b \in S : \; a \overset{v}{\simeq} b \wedge a \overset{w}{\simeq} b \implies a = b$$
- negates weights when operands are swapped:
$$\forall a, b \in S : \; a \overset{w}{\simeq} b \implies b \overset{-w}{\simeq} a$$

Valid weighted preorder

Definition (Valid weighted preorder)

A weighted preorder is *valid* when weights are bouded:

$$\forall a, b \in S, \exists w_0 \in \mathbb{Z} : \; a \overset{w}{\precsim} b \implies w \geq w_0$$

Proposition

A valid weighted preorder induce a weighted equivalence:

$$a \overset{w}{\precsim} b \wedge b \overset{-w}{\precsim} a \iff a \overset{w}{\simeq} b$$

Static partitioning

Definition (Static partitioning)

A static partitioning is a valid weighted preorder that:

- contains dependencies

$$b(t) \text{ depends on } a(t-w) \implies a \overset{w}{\precsim} b$$

- maps dependencies on inputs and outputs

$$i \overset{w}{\precsim} o \implies o(t) \text{ depends on } i(t-w)$$

Theorem

Static partitionings are exactly the grey-boxings that do not reject any calling context and respect dependencies.

How to find a mimimal grey-boxing

Encode in the quantifier-free Presburger arithmetic:

$\precsim$ is a static partitioning with k classes

Use a SMT solver

- Try to satisfy the formula for $k = 1, 2, 3, \ldots$
- Stop when a solution is found
- Exponential complexity, but the problem is NP-Hard

Heuristic

Preorder saturation

- The dependency relation $a \xrightarrow{w} b$ is a static partitioning
  - each group contains a single equation: full inlining
- Add constraints to the weighted preorder to form groups

Arguments in favor of the heuristic

- Based on a heuristic that does not handle retiming
  [Pouzet and Raymond 2009]
- Optimal on inputs and outputs
- A sub-optimal partitioning is still better than inlining

Outline

Destructive updates

Modular scheduling

Scheduling algorithms

Conclusion and future work

Highlights

Destructive updates for synchronous dataflow languages

- Array updates are in-place by default
- No copies between reactions

Modular retiming

- Allows inter-reaction scheduling
- Gives more flexibility
- Other uses ?

Scheduling constraints to enforce destructive updates

- A typing system is traditionally used instead
- Could be applied to conventional functional languages

Future work

Aliasing analysis

```
(* Do A and B alias ? *)
A: int[8] = if x then X else Y
B: int[8] = if x then Y else X

(* Do A[i] and A[C[j]] alias ? *)
C = A[i] <- 0
x = A[C[j]]
```

Memory management

- Multiple array location possible per variable
- Live range of arrays are unknown

Handle clocks