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

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

\[ a \rightarrow b \rightarrow c \]
Equation Ordering

```plaintext
node foo(a: int) = (c: int) {
    c = b + 1 (* ct = bt + 1 *)
    b = a + (0 -> pre b) (* b0 = a0 *)
    (* bt = at + bt-1 *)
}
```

Implicit ordering of equations given by data dependencies

- Ordering in source code doesn’t matter

Allow feedback loops

- Every dependency loop must cross a delay
Compilation to efficient C code

\[
\text{node } \text{foo}(a:\ \text{int}) = (c:\ \text{int}) \{
\begin{align*}
    c &= b + 1 \quad (* c_t = b_t + 1 *) \\
    b &= a + (0 \rightarrow \text{pre } b) \quad (* b_0 = a_0 *) \\
        &\quad (* b_t = a_t + b_{t-1} *)
\end{align*}
\}
\]

Generated C code

// Computes a time step of foo
\[
\text{void} \ \text{foo}_\text{step}(\text{foo}_\text{state}_t* \text{state}, \text{int} \ a, \text{int}* \ c) \{
\begin{align*}
    \text{int} \ b &= a + ((t == 0) ? 0 : \text{state}->b); \\
    *c &= b + 1; \\
    \text{state}->b &= b; \quad // \text{Update state}
\end{align*}
\}
\]
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 \cdot 1000  
f = e[4]  
  ```

- Define a new array from an old one
  
  ```
  g = e[3] <- 42  
  ```

Immutable arrays: no in-place update

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

int g[1000];  
memcpy(g, e, 1000*\texttt{sizeof(int)});  
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 *)

\[
b = a[0] \leftarrow 0
\]

(* Access a *)

\[
c = a[0]
\]

Copies are no longer needed

- Modifications of the original array cannot be observed
- Reuse the memory of the original array
Strengths 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 \texttt{pre}? Two solutions:

1. Insert a copy at every \texttt{pre}
2. \textbf{Handle inter-reaction aliasing}

\begin{align*}
(* \ b_t \ consumes \ a_t \ *) \quad & \\
\texttt{b} &= \texttt{a}[0] \ <- \ 0
\end{align*}

\begin{align*}
(* \ c_{t+1} \ accesses \ a_t \ *) \\
\texttt{c} &= (\texttt{pre} \ \texttt{a})[0]
\end{align*}

\texttt{c}_{t+1} \ must \ be \ executed \ before \ \texttt{b}_t

\textbf{Must \ \textit{retine} equations}

1. 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
node f(A, B: int[8])
    = (c: int) {
      D = B[0] <- 0
    }

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

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

Is there a dependency from Read A to Write B?
node f(A, B: int[8])
   = (c: int) {
      D = B[0] <- 0
   }

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

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

Exposé reads and writes to the calling context
node f(A, B: int[8])
    = (c: int) {
        D = B[0] <- 0
    }

(* 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:

\[ \forall t \in \mathbb{N} : a_t \text{ depends on } b_{t-w} \quad \text{with } w \in \mathbb{Z} \text{ constant.} \]

Dependencies represented by a weighted graph:

```plaintext
node foo(a, b: int) = (c: int) {
    (* c_t = a_t + b_{t-1} *)
    c = 0 -> a + pre b
}
```
How can we schedule this node?
{a_{t-1}, b_t} before {d_t, e_t}
Modular Compilation - Feedback Loops

\[
\begin{align*}
\{d_t, e_t\} & \text{ before } \{a_{t-1}, b_t\}
\end{align*}
\]
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: 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 \( r_i : X_i \rightarrow \mathbb{Z} \)
  - \( a_t \) is computed at the reaction \( t + r(a) \)
- A dependency relation \( X \xrightarrow{w} Y \) on sub-nodes
  - \( X \xrightarrow{w} Y \Rightarrow Y_t \) 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 exiting scheduling algorithms
Destructive Updates for Synchronous Dataflow Languages

With copying \texttt{pre}

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

With inter-iteration aliasing

- \texttt{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