Electronic Notes in Theoretical Computer Science 65 No. 5 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume65.html 8 pages

Efficient compilation of array iterators for Lustre

Lionel Morel

Verimag - Institut National Polytechnique de Grenoble (INPG)
Centre Equation - 2 Avenue de Vignate, 38610 GIERES, France

Abstract

In this paper, we present a compilation scheme along with optimizations for array iterators in the synchronous data flow language Lustre. We present the iterators (inspired from functional languages mechanisms such as *map* and *foldl*...) and show how to compile them into efficient imperative code that manipulates arrays and loops instead of inefficient code were arrays are expanded into independent variables.

1 Introduction

Lustre[4] is a synchronous data flow language developed at Verimag since 1984. It is dedicated to the programming of reactive systems particularly used in critical applications (power plants, airplanes, etc.). It is the core language of the *Scade* graphical tool developed by *Esterel Technologies* and used for developing control systems for *Schneider*'s power plants and *Airbus*.
The language provides constructors powerful enough to describe every feature of such systems and restrictive enough to answer criticality constraints: we don't want any dynamical error (bound overflow, invalid pointer access, etc.).
Arrays have been introduced in Lustre-V4 for hardware purposes (mainly initialization of regular circuits[5]) for which the operators proposed (slice extractions, concatenation of arrays) were particularly adapted. As we shall see, the compilation techniques used are adapted to hardware development and to verification, but not to software development (the main application of Lustre): the code produced is not efficient.
In order to solve these problems, some *iterators* were proposed by N. Halbwachs and F. Maraninchi, adapting to Lustre constructs widely used in functional programming[3]. The idea was to give constructs for which it is always easy to generate efficient loop code.
Section 2 presents some concepts about Lustre. Section 3 presents the Lustre iterators. Sections 4 and 5 present compilation and optimization algorithms

to generate efficient imperative code for the iterators. Section 6, presents implementation and case-study results. Section 7 concludes and gives the main perspectives of this work.

2 The Lustre language and programming environment

Lustre[4] is a data-flow (every variable or expression denotes a sequence of values) and synchronous (programs are ruled by a clock) language. Its style is declarative: outputs of programs are defined by unordered equations.

2.1 Overview of the language

Variables, expressions and equations: Every variable or expression denotes a flow, i.e. a possibly infinite sequence of values of a given type. A variable `X` represents the sequence $x_0, x_1, \ldots, x_i, x_{i+1}, \ldots$ (x_i being the value of x at the i-th instant of the execution of the program). Basic types (`boolean`, `integer` and `real`) can be organized in tuples and arrays. Any operator from $\tau_1 \times \tau_2 \times \ldots \times \tau_n$ to τ (τ and τ_i being types) is applied pointwise to infinite sequences of these types. Thus if $X = (x_1, x_2, \ldots, x_i, \ldots)$ and $Y = (y_1, y_2, \ldots, y_i, \ldots)$ then $X + Y = (x_1 + y_1, x_2 + y_2, \ldots, x_i + y_i, \ldots)$. Usual arithmetic (`+`, `-`, `*`, `/`, `div`, `mod`), boolean (`or`, `and`, `not`), relational (`=`, `<`, `<=`, `>`, `>=`) and conditional (`if...then...else...`) operators are provided.
Temporal operators: There exists an operator `pre` to talk about the past of an expression: $\forall n > 0$, $\texttt{pre}(X)_n = x_{n-1}$. So $\texttt{pre}(X) = (\texttt{nil}, x_1, x_2, \ldots, x_i, \ldots)$ (`nil` denotes an undefined value). The initialization operator $\rightarrow$ is also provided: $X \rightarrow Y{=}(x_1, y_2, \ldots, y_i)$.
Equations and nodes: A variable `id` is defined by an equation: `id = expression`. Equations can be grouped in *nodes* that have *inputs*, *outputs*, *local variables* and a *set of equations* defining the outputs and local variables. Nodes can be connected with each other by node calls.

2.2 Constraints on cyclic definitions and compilation

A set of equations describes a network of operators, and is equivalent to the description of a combinational circuit. The same constraints apply: sets of equations with instantaneous loops (like : $\{x = y + z, z = x + 1, ...\}$) are ruled out by the compiler (it is a set of fix-point equations that perhaps has solutions, but it is not accepted as a dataflow program). Lustre programs are compiled into imperative programs in C, which have the following form:

```
<Initialize memories>
Always do{
  <Read inputs>
  <Compute outputs>
  <Update memories>}
```

```
node N(Y : int^6) returns (X : int^6);
let
  X[1..2] = Y[4..5] ;
  X[3..5] = Y[1..3] ;
  X[0] = Y[0] ;
tel
```

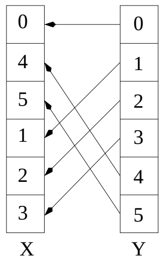

Fig. 1. A slice expression and the corresponding dependencies between `X` and `Y`

2.3 Manipulating arrays in Lustre-V4

Arrays have been introduced in Lustre-V4 for helping the description of regular circuits [5]. The operators and associated compilation techniques are specially designed for this kind of development.

If τ is a type and `n` is a constant (known at compile-time: for criticality reasons we don't allow array access by dynamic indices), then τ`^n` is the type of arrays of size `n` and whose elements' type is τ. The following constructs are provided: `[0,2,3]` is the array of elements 0,2 and 3; `true^3` is the array (`[true,true, true]`); slice extraction: $\mathtt{A[i..j]} = [\mathtt{A[i]}, \mathtt{A[i+1]}, \ldots, \mathtt{A[j]}]$ (if $\mathtt{i} \leq \mathtt{j}$) or $[\mathtt{A[i]}, \mathtt{A[i-1]}, \ldots, \mathtt{A[j]}]$ (if $\mathtt{j} < \mathtt{i}$); Concatenation : $\mathtt{A|B} = [\mathtt{A[0]}, \mathtt{A[1]}, \ldots, \mathtt{A[n-1]}, \mathtt{B[0]}, \mathtt{B[1]}, \ldots, \mathtt{B[m-1]}]$ All the polymorphic operators (`if...then...else...`, `pre`, $\rightarrow$) are extended to arrays.A static recursion mechanism is also available.

Compilation with expansion: At compilation, arrays are expanded into independent variables. When we use an array of n elements in a Lustre node, we obtain n independent variables. This technique is necessary when programming hardware systems in which data arrangement has no meaning anyhow, and for verifying programs (since we don't know how to take advantage of the data arrangement for proving properties) but it doesn't fit software efficiency requirements. For formal verification purposes it's unfortunate not to use the information given by the data arrangement to help proving properties of the system (see section 7).

Compilation without expansion: Lustre-V5 was an attempt to generate loop code from the operators presented above. But, using slice mechanisms, one can write a program like the one given in figure 1. It is clearly tedious (even though possible) to write an equivalent imperative loop for this kind of program.

Conclusion: 1) compilation techniques presently used for arrays are not adapted for obtaining efficient code; 2) it is not always easy to generate efficient loop imperative code from the operators provided in the language (slice expressions); 3) these operators are not easy to use when programming classical array algorithms (sorting, maximum, etc.) 4) when expanding arrays into independent variables data arrangement is lost while it could be kept for verification.

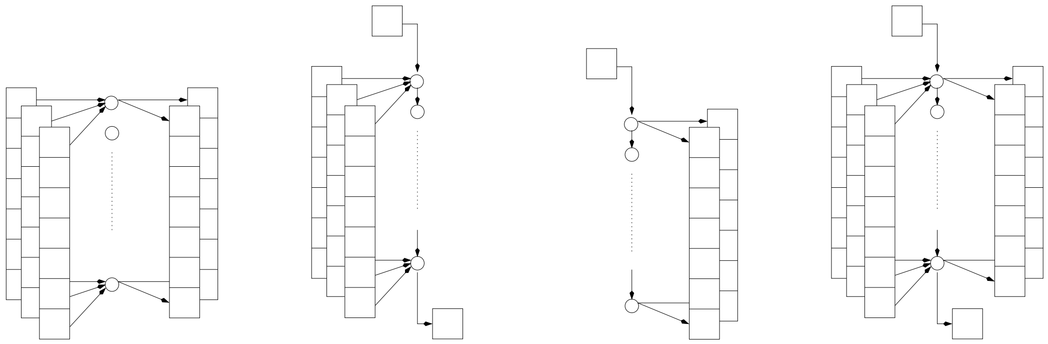

Fig. 2. The 4 iterators `map`, `red`, `fill` and `map_red`

3 Iterations in Lustre

Our work introduce iterators inspired from functional operators like *map* or *foldl* into Lustre. They only allow simple dependences between array elements and thus make easier the generation of loop-code. Generating loop presents the following advantages considering the code generated: 1) *size*: In all the cases where we apply n times a computation C, we reduce the number of copies of C from n to 1; 2) *execution time*: a C program containing n assignments written in sequence is generally a bit slower than an equivalent program with a loop containing 1 assignment executed n times; 3) *amount of memory needed*: If we use iterators it is possible to identify and suppress useless intermediate variables (see section 5); 4) *readability*: the operators we propose are easy to manipulate (their use is similar to intuitive functional operators).
Iterators have been widely used in functional programming for more than twenty years [2]. Operators such as *map*, *foldr* and *foldl* (*reductions*) are very well defined in [3]. Attempts were made to introduce such data-parallelism techniques in non-purely functional specification languages (in $8_{1/2}$[6], Alpha[1] or ID[7]). Our contribution consists mainly in adapting these constructs to a data-flow *synchronous* language. Safety constraints have an important influence on the constructs we introduce and on how to compile them.
In the sequel, n is an integer whose value must be know statically. T and T' are arrays of size n. The τs are types and $\tau\hat{}n$ is the type "*array of size n of elements of type* τ". The size of the arrays are necessary only for the `red` operator, but for uniformity we give it even for the others. For simplicity, definitions (and optimizations given in section 5) are only given for iterations of purely functional nodes. Figure 2 gives a graphical description of the iterators.

`map`: If $g = \lambda t.t'$, where t represents an array element and t' an expression depending on t, an abstract syntax of the `map` operator is : $T' = map(g, T)$. It is semantically equivalent to: $\{T'[i] = g(T[i])\}_{i \in \mathrm{range}(T)}$. If `N` (resp. `O`) is a node (resp. an operator) of signature $\tau_1 \times \tau_2 \times \ldots \times \tau_l \to \tau'_1 \times \tau'_2 \times \ldots \times \tau'_k$, then `map<<N,n>>` (resp. `map<<O,n>>`) is a node (resp. an operator) [1] of signature:

[1] from now on, we will not make the distinction between nodes and operators

```
node memo(accu_in : int)
  returns (accu_out, t : int);
let
  accu_out = 0 -> pre(accu_in);
  t = accu_in;
tel

node Tenlast(V : int)
  returns (T : int^10);
let
  T = fill<<memo,10>>(V);
tel
```

```
int _V;
int _T[10];
int PRE_accu_in[10];
int _accu_out;
int _accu_in[10];
//Initializing
// of iteration
_accu_out = V;
```

```
//Computing outputs (_V)
for(i=0;i<10;i++){
 _T[i] = _accu_in[i];
 if(init){ _accu_out = 0;}
 else{
   _accu_out = PRE_accu_in[i];}}
//Memorizing values
for(i=0;i<10;i++){
   PRE_accu_in[i] = _accu_in[i];}
```

Fig. 3. A Lustre program manipulating iterators along with the corresponding C code

$\tau_1\hat{}\mathrm{n}\times\tau_2\hat{}\mathrm{n}\times\ldots\times\tau_l\hat{}\mathrm{n}\rightarrow\tau_1'\hat{}\mathrm{n}\times\tau_2'\hat{}\mathrm{n}\times\ldots\times\tau_k'\hat{}\mathrm{n}$

`red`: If $g = \lambda(t, accu).accu'$, the reduction r of an array T using g is : $r = red(init, T, g)$, where $init$ is the initialization expression of the reduction. It is semantically equivalent to: $\{r_0 = init; \{r_{i+1} = g(r_i, T[i])\}_{i\in range(T)}; r = r_{size(T)}\}$. The operator `red` has this syntax: If `N` is a node of signature: $\tau\times\tau_1\times\tau_2\times\ldots\times\tau_l\rightarrow\tau'$ then `red<<N,n>>` is a node of signature: $\tau\times\tau_1\hat{}\mathrm{n}\times\tau_2\hat{}\mathrm{n}\times\ldots\times\tau_l\hat{}\mathrm{n}\rightarrow\tau'$.

`fill`: If $g = \lambda accu.(accu', elt)$, we can have $T = fill(init, g)$, where $init$ is the initialization of the filling process. It is semantically equivalent to $\{r_0 = init; \{r_{i+1}, T[i] = g(r_i)\}_{i\in range(T)}\}$. `fill` has this syntax: If `N` is a node of signature $\tau\rightarrow\tau\times\tau_1'\times\tau_2'\times\ldots\times\tau_k'$ then `fill<<N,n>>` is a node of signature $\tau\rightarrow\tau_1'\hat{}\mathrm{n}\times\tau_2'\hat{}\mathrm{n}\times\ldots\times\tau_k'\hat{}\mathrm{n}$.

`map_red`: If $g = \lambda(accu, t).(accu', t')$ we have $(T', r) = map\_red(init, T, g)$. It is semantically equivalent to $\{r_0 = init; \{r_{i+1}, T'[i] = g(r_i, T[i])\}_{i\in range(T)}; r = r_{size(T)}\}$. In Lustre, if `N` is a node of signature $\tau\times\tau_1\times\tau_2\times\ldots\times\tau_l\rightarrow\tau\times\tau_1'\times\tau_2'\times\ldots\times\tau_k'$, then `map_red<<N,n>>` is a node of signature $\tau\times\tau_1\hat{}\mathrm{n}\times\tau_2\hat{}\mathrm{n}\times\ldots\times\tau_l\hat{}\mathrm{n}\rightarrow\tau\times\tau_1'\hat{}\mathrm{n}\times\tau_2'\hat{}\mathrm{n}\times\ldots\times\tau_k'\hat{}\mathrm{n}$.

Extensions: With these iterators, it is not obvious to write, e.g., a transposition. To extend their power of expression, we propose two other operators: `transpose` and `mirror`. If T is an expression whose type is $\tau\hat{}n\hat{}m$, the `transpose(`T`)` is an expression of type $\tau\hat{}m\hat{}n$ defined by: $\forall k, l \in [0, n-1]\times[0, m-1]$, `transpose(`$T$`)`$[k, l] = T[l, k]$ for `mirror`, If T is an expression of type $\tau\hat{}n$. Then, `mirror(`T`)` is an expression of type $\tau\hat{}n$ defined by: $\forall\ k \in [0, n-1]$, `mirror(`T`)`$[k] = T[n-k-1]$.

4 Compiling a simple example

Compilation of iterators is based on techniques developed in [12]. First, let's see an example. We want to store the 10 last values of a variable `V` in an array

```
node main(T : int^10)returns(T'' : int);
var T' : int^10;
let
  T' = map<<f,10>>(T); T'' = map<<g,10>>(T');
tel

node f(in_f : int)returns(out_f : int);
let
  out_f = in_f + 1;
tel
node g(in_g : int)returns(out_g : int);
let
  out_g = in_g + 2;
tel
```

```
node main(T : int^10)returns(T'' : int);
let
   T'' = map<<h,n>>(T );
tel

node h(in_f : int)returns(out_g : int);
var
   in_g : int ;
   out_f : int ;
let
   out_f = in_f + 1; in_g = (out_f);
   out_g = in_g + 2;
tel
```

Fig. 4. An example of cascade (left) and the same program optimized (right)

`T`. The Lustre program is written in figure 3 (left).
The code we can generate for such a program contains arrays for inputs and outputs of iterations along with arrays for memories corresponding to the `pre`s in the node iterated. We can distinguish three parts as in any synchronous program: 1) Memories are initialized; 2) Outputs are computed: a loop over arrays for memories and for inputs; 3) Values associated to the various `pre`s are memorized for the next instant (this part is also written as a loop over arrays). For generating the "memorizing" code we use classical techniques originally proposed in [8] (see figure 3).

5 Optimization

A possible optimization appears when writing cascades of iterations. Consider the program presented on figure 4 (left). We replace it by the one given on figure 4 (right), `h` being a new node, composition of `f` and `g` (assuming that `T'` is not used by the rest of the program). This kind of optimization is known in functional programming as *listlessness* [9,10] or *deforestation* [11]. We use listlessness principles: instead of generating the whole array `T'`, its elements are consumed as soon as they are produced. This optimization will be very useful in a context where programmers manipulate libraries of nodes performing classical array algorithms (e.g. in *Scade*) thus not necessarely knowing that cascades appear.
Axiomatization: In order to apply the optimization axioms, we must have that: 1) The result(s) of the first iteration are the input(s) of the second one 2) These variables are not used in the rest of the node (local variables only appearing in the two iterations) 3) The cascade formed by the two iterations is optimizable (see figure 5)

For every optimizable cascade there is an axiom. E.g.: If $f = \lambda a, t.(a', t')$ and $g = \lambda a, t.(a'', t'')$ (where a', t', a'' and t'' depend on a and t), if i_2 does not depend on r_1, we can apply the rule given on figure 6.

↓ followed by →	map	fill	red	map-red
map	⋆		⋆	⋆
fill	⋆		⋆	⋆
red				
map-red	⋆		⋆	⋆

Fig. 5. The optimizable cascades

$$
\begin{array}{c}
map\_red(i_2, map\_red(i_1, T, f), g) \\
\equiv \\
map\_red(\{i_1, i_2\}, T, \lambda\{a_f, a_g\}, t\ . \\
let < x, y > = f(a_f, t) \\
in\ let < x', y' > = g(a_g, y) \\
in < \{x, x'\}, y' >).
\end{array}
$$

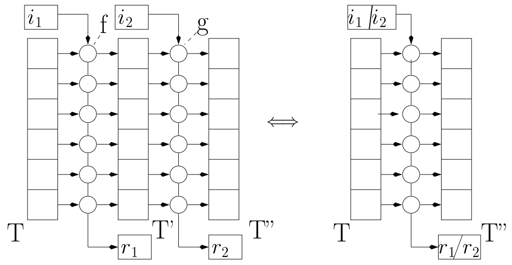

Fig. 6. a `map_red` followed by a `map_red`

6 Prototyping and applications

The iterators have been implemented in a prototype compiler that can generate C code for simple Lustre programs that manipulate iterations. They were not introduced in the academic compiler: we wanted to experiment our approach first. We treated several kinds of classical algorithms that use arrays (sorts, maximums, etc.) and showed that the code obtained is more efficient than the one obtained for an equivalent program in Lustre-V4. We also applied our proposition to a real-scale case study provided by Airbus that contains approximately 350 Scade nodes with lots of arrays. This showed that: 1) iterators are really useful: lots of programs written were actually iterations but performed on expanded arrays with duplication of the code; 2) the optimizations we propose could be apply very frequently in the whole program: developers did write lots of computations that can be seen as cascade of iterations. Iterators will soon be added to Scade and used by Airbus.

7 Conclusion and further work

New operators for manipulating arrays in the data-flow synchronous language Lustre have been proposed. These operators, inspired by the ones generally used in functional programming were designed such that we can easily generate efficient imperative code. We proposed optimizations made at the source level in order to provide more efficient code: 1) memory normally used for the intermediate arrays is saved; 2) the program produced is faster than before optimizations (we suppress two loops: one that usually writes the intermediate array and an other one that reads it). The iterators were implemented in a prototype compiler which validated the compilation process. These constructs will be added to the academic version of Lustre and to Scade. Programers will use libraries built with the iterators and will not necessarily know that optimizations are needed or applied. Moreover, we think it is possible to use the arrangement of data to help proving properties. For example, if $B = map(f, A)$, we have an invariant property of B ($B[i]$ depends on $A[i]$). A theorem prover (like PVS) can be used to prove these invariants. We are currently studying this last question.

References

[1] Equipe API-Cosi. *Getting started with Alpha.* IRISA, September 1997.

[2] J. Backus. Can programming be liberated from the von neuman style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[3] R. S Bird. Lectures on constructive functional programming. In *Constructive Methods in Computer Science*, pages 151–218. Springer-Verlag, 1988.

[4] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proc. of the IEEE*, 79(9):1305–1320, September 1991.

[5] N. Halbwachs and F. Rocheteau. Implementing reactive programs on circuits: A hardware implementation of LUSTRE. In *Proc. of Real-Time: Theory in Practice*, volume 600 of *LNCS*, pages 195–208, Berlin, Germany, June 1992. Springer.

[6] O. Michel, D. De Vito, and J-P Sansonnet. 8_1/2 : Data-parallelism and data-flow. Technical report, LRI-CNRS, Université Paris-Sud, 1992.

[7] R. S. Nikhil. *ID Reference Manual.* MIT Laboratory for Computer Science, 90.1 edition, July 1991.

[8] P. Raymond. *Compilation efficace d'un langage déclaratif synchrone : le générateur de code Lustre-V3.* PhD thesis, Institut National Polytechnique de Grenoble, November 1991.

[9] P. Wadler. Listlessness is better than laziness. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 45–52. ACM, August 1984.

[10] P. Wadler. Listlessness is better than laziness II: composing listless functions. In *Proc. Workshop on Programs as Data Objects*, volume 217 of *LNCS*. Springer Verlag, 1985.

[11] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *Proc. of the European Symposium on Programming*, volume 300 of *LNCS*, pages 344–358. Springer Verlag, 1988.

[12] R. C. Waters. Automatic transformation of series expressions into loops. *ACM Transactions on Programming Languages and Systems*, 13(1):52–98, January 1991.