Polyhedra at Work : Automatic Generation of VHDL Code for the Sherman-Morrison Formula

Michel Lemaire[1], Daniel Massicotte[1], Jeremy Poupart[1], Patrice Quinton[2] and Sanjay Rajopadhye[3]

[1]Université du Québec à Trois-Rivières
[2]École normale supérieure de Rennes
[3]Colorado State University

Impact 2024, January 17, 2024

Introduction

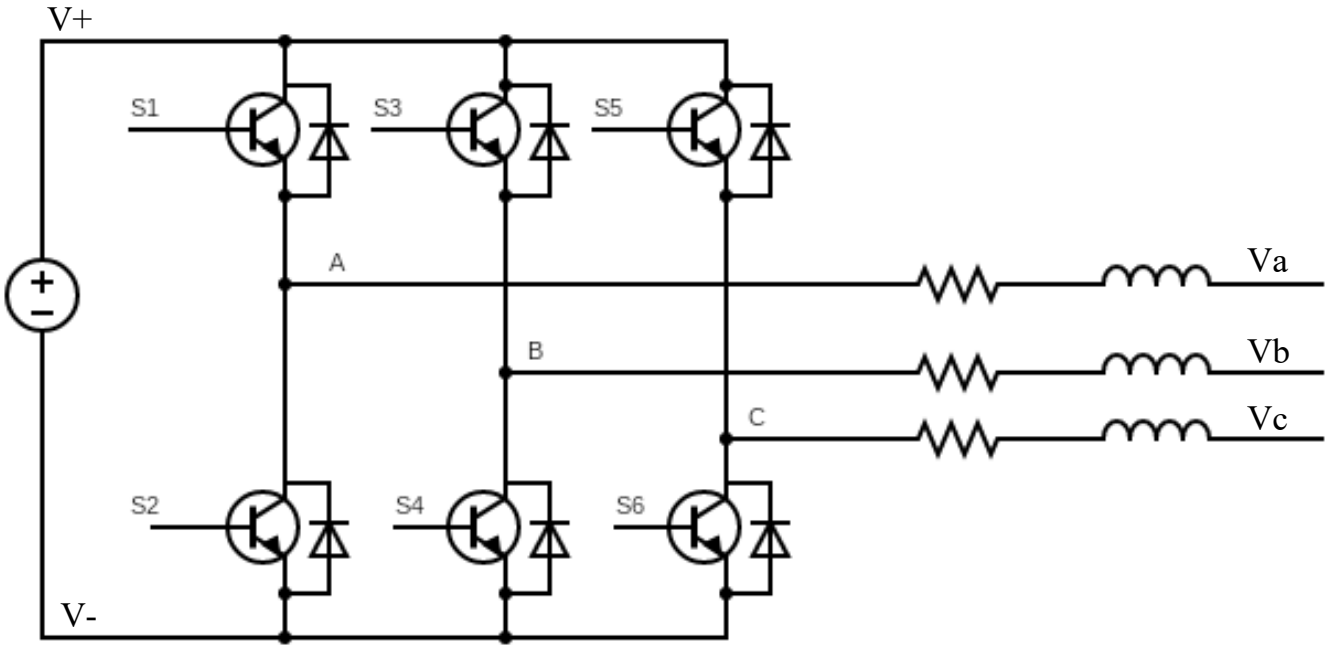
- Context : simulation of electrical circuits including switches
- Real-time, low latency requires FPGA, parallel implementation
- Use of the Polyhedral equational model (ALPHA language)
- Automatic generation of synthesizable VHDL for Sherman-Morrison based algorithms
- VHDL code targeted to the System Generator software
- Experiment the design flow on two versions of Sherman-Morrison

Outline

- Electrical Simulation
- ALPHA : A Polyhedral Equational Language
- Synthesis steps
- Results and Discussion
- Conclusion

Simulation of Electrical Circuits

A Power-Converter



Admittance Matrix and Simulation

$$A = \begin{bmatrix}
g1+g3+g5 & 0 & -g1 & -g3 & -g5 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & g2+g4+g6 & -g2 & -g4 & -g6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
-g1 & -g2 & g1+g2+gR1 & 0 & 0 & -gR1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-g3 & -g4 & 0 & g3+g4+gR2 & 0 & 0 & -gR2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-g5 & -g6 & 0 & 0 & g5+g6+gR3 & 0 & 0 & -gR3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -gR1 & 0 & 0 & gL1+gR1 & 0 & 0 & -gL1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -gR2 & 0 & 0 & gL2+gR2 & 0 & 0 & -gL2 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -gR3 & 0 & 0 & gL3+gR3 & 0 & 0 & -gL3 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -gL1 & 0 & 0 & gL1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -gL2 & 0 & 0 & gL2 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -gL3 & 0 & 0 & gL3 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

- Admittance : inverse of resistance
- Simulation : solve $Ax = B$ for x, where B is the vector of currents and voltages
- Finding out A^{-1} is needed

Summary of the Problem

- Simulation based on repeated computation of $x = A^{-1}Bb$, where A^{-1} is the inverse of admittance matrix A
- Matrix A changes when switch states are modified
- Calculation of A^{-1} needed
- Direct methods have $O(N^3)$ complexity
- Use perturbation methods (here, Sherman-Morrison)

The Sherman-Morrison Formula

- The Sherman-Morrison formula :

$$\begin{aligned}(A + uv^T)^{-1} &= A^{-1} - \sigma A^{-1} uv^T A^{-1} \\ \sigma &= \frac{1}{1 + v^T A^{-1} u}\end{aligned}$$

- Complexity is $O(N^2)$
- Repeat the use of the formula when several switches change
- For n switches, 2^n different A^{-1} matrices
- The method can be used from any A^{-1} matrix, with the appropriate u and v vectors.

Alpha : A Polyhedral Equational Language

```
1   system matVect: {N | 2<=N}
2     (a : {i,j | 1<=i<=N; 1<=j<=N} of integer;
3      v : {i | 1<=i<=N} of integer)
4   returns (c : {i | 1<=i<=N} of integer);
5   var
6     X : {i,j | 1<=i<=N; 0<=j<=N} of integer;
7   let
8     X[i,j] =
9         case
10          { | j=0} : 0;
11          { | 1<=j} : X[i,j-1] + a[i,j] * v[j];
12        esac;
13    c[i] = X[i,N];
14  tel;
```

Transformations

- Substitution, and anti-substitution (allows reformating of code)
- Normalization (simplification of expressions, thanks to a set of axiomatic rules)
- Change of basis (to perform time-space mapping, after scheduling)
- Pipelining (to organize and simplify the communications)
- Simplification of reductions (to reduce the complexity of the code)
- Tiling

Tools : MMAlpha and alphaz

- MMAlpha (Irisa) : "compiler" that targets the automatic generation of hardware.
- Includes :
  - A scheduler
  - Various simulators
  - A VHDL translater
  - Allows structured design
- alphaz (CSU) : "transformation explorator framework"
- Includes
  - Tiling
  - Memory mapping of variables
  - Simplification of reductions
  - Parallel Code generation
- Syntax differs slightly, but abstract syntax is identical

ALPHA code for the Sherman-Morrison Formula (1/2)

```
1   -- B represents A^-1
2   system shermanMorrison: {N | 2<=N}
3   (
4     B : {i,j | 1<=i<=N; 1<=j<=N} of integer;
5     u : {i | 1<=i<=N} of integer;
6     v : {i | 1<=i<=N} of integer
7   )
8   returns
9   (
10    newB : {i,j | 1<=i<=N; 1<=j<=N} of integer
11  );
12  var
13    Btrans: {i,j | 1<=i<=N; 1<=j<=N} of integer;
14    oprv, incrA : {i,j | 1<=i<=N; 1<=j<=N} of integer;
15    sigma: integer;
16    r: {i | 1<=i<=N} of integer;
17    l: {i | 1<=i<=N} of integer;
18    d: integer;
```

Alpha code for the Sherman-Morrison Formula (2/2)

```
19  let
20    use transpose[N] ( B ) returns ( Btrans );
21    use matVect[N] ( B, u ) returns ( r );
22    use matVect[N] ( Btrans, v ) returns ( l );
23    use dot[N] ( l, u ) returns ( d );
24    sigma[] = 1[];
25    use outProd[N] ( r, l ) returns (oprv);
26    incrA[i,j] = sigma[]*oprv[i,j];
27    newB = B - incrA;
28  tel;
```

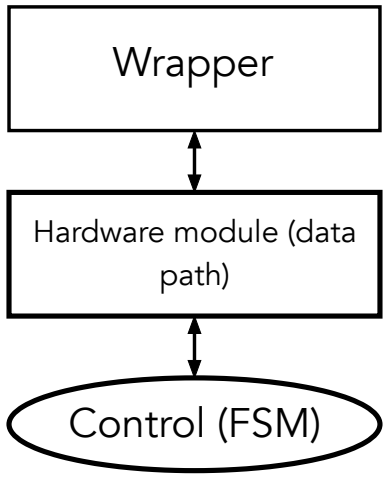
Synthesis Steps in MMAlpha

- Parse, and type-check
- Inline subsystems
- Schedule (using the *vertex method*)
- Time-space map (using change of basis transformation)
- Transform to multi-dimensional RTL code (in Alpha)
- Translate to VHDL

Until the translation to VHDL, the program is expressed in Alpha, and is strictly equivalent to the initial program.

Translation to VHDL

Separation of code (structured ALPHA)



- Wrapper : interface, I/Os, generation of stimuli files
- Hardware module : translated as a Data-path
- Controller : translated as a finite state machine

Translation of the Hardware module

- Equation by equation translation
- Target : multi-dimensional, synthesizable, RTL subset of VHDL
- Variables are mapped on VHDL arrays of signals
- Combinational equations : `X[t,p] = f(Y[t,p], …)`
- Simple connections : `X[t,p] = Y[t,p]`
- Multiplexers (using `if` and `case` expressions)
- Register equations : `X[t,p] = Y[f(t,p),g(t,p)]`
- Control equations, involving boolean variables (in the controller)

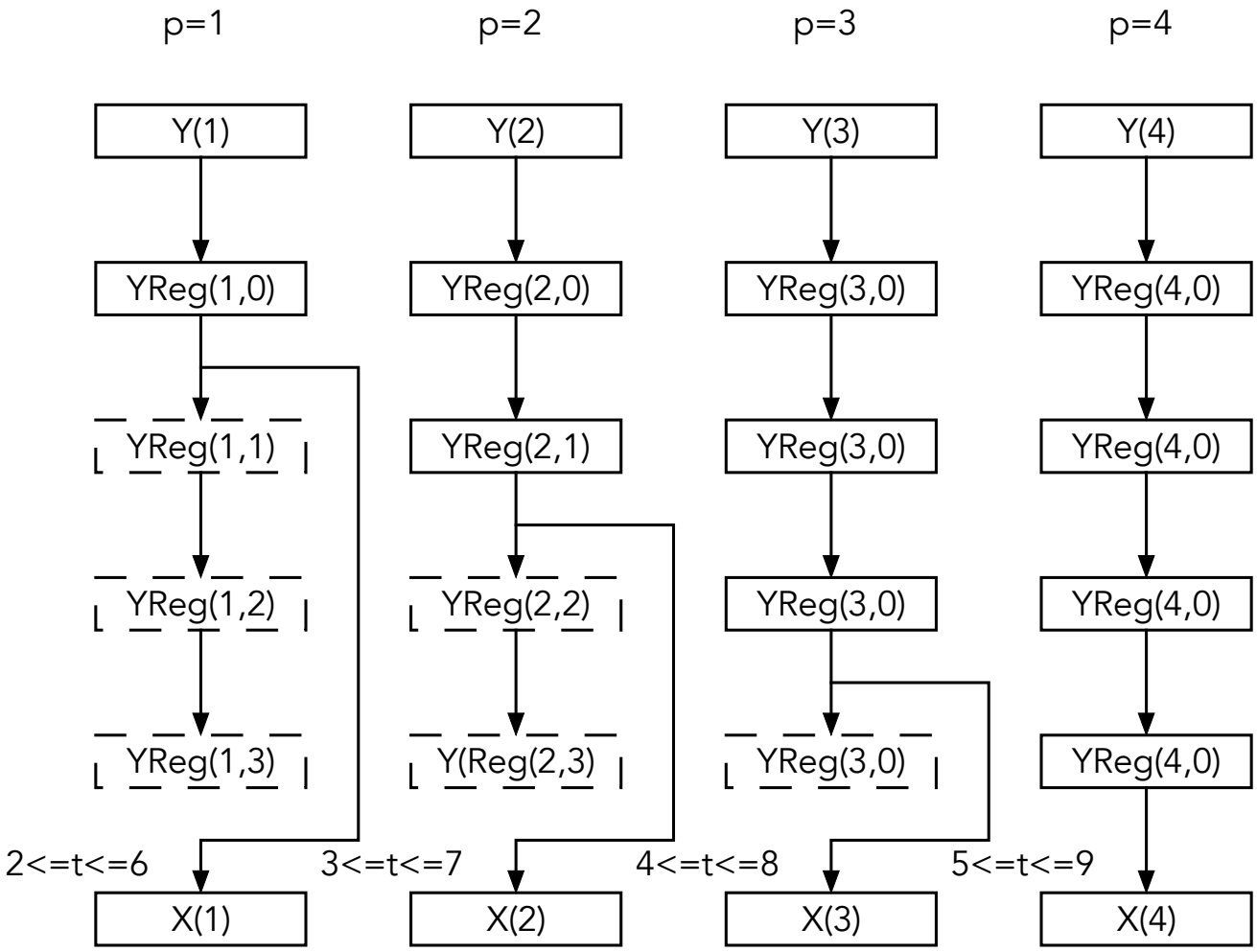
Translation of *register equations*

- General form : `X[t,p] = Y[f(t,p),g(t,p)]`
- Life-time analysis of variable `Y` using PIP (Parameter Integer Programming)
- Declare, if needed, an additional register variable to store `Y` as long as needed
- Generate VHDL by separating the *time* and the *space* dependency
- There is room for optimizations...

Translation of register equations (example)

```
1     -- X[t,p] = Y[t-p,p]
2     -- Domain: {t,p | p+1<=t<=p+4; 1<=p<=4}
3     G81: FOR p IN 1 TO 4 GENERATE
4       G79: FOR i IN 1 TO 4 GENERATE
5         -- Time dependence
6         PROCESS(clk) BEGIN
7           IF (clk = '1' AND clk'EVENT) THEN
8             IF CE = '1' THEN
9               YReg(p)(i) <= YReg(p)(i-1);
10              YReg(p)(0) <= Y(p);
11            END IF;
12          END IF;
13        END PROCESS;
14      END GENERATE;
15      -- Spatial dependence
16      X(p) <= YReg(p)(-1 + p) WHEN (counter - counterDelay >= p+1)
17        AND  (counter - counterDelay <= p+4);
18    END GENERATE;
```

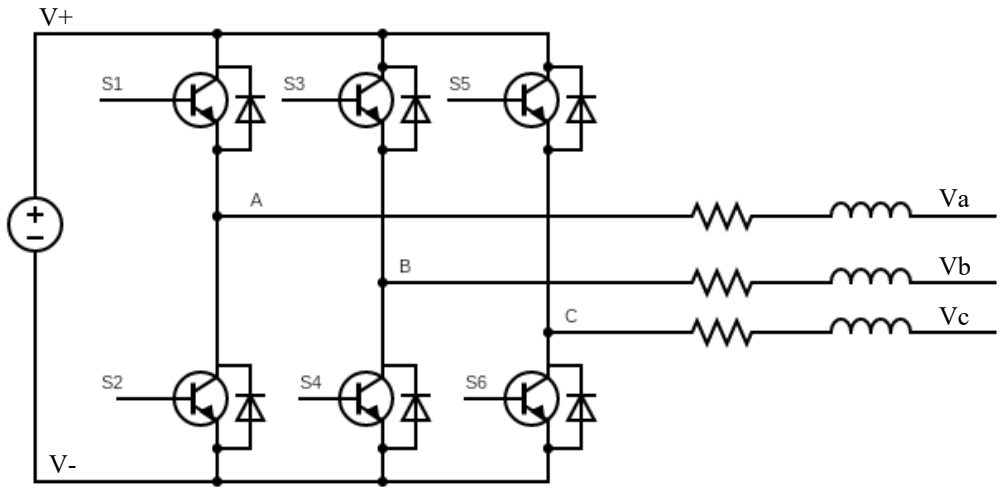
Translation of *register equations* (exemple)



Optimized Code for the Electrical Simulation

S1 switch

$$\begin{pmatrix} g1 + g3 + g5 & 0 & -g1 \\ 0 & g2 + g4 + g6 & -g2 \\ -g1 & -g2 & g1 + g2 + gR1 \end{pmatrix}$$



Optimized Version

Change of a switch status

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & -g1 & 0 & g1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & g1 & 0 & -g1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

can be represented as uv^T, where $u = (0, g1, 0, -g1, 0)$ and $v = (0, -1, 0, 1, 0)$ (non-zero elements in positions i and j).

Optimized version

To compute $A^{-1} \leftarrow A^{-1} + \sigma A^{-1} u v^T A^{-1}$:

- Select columns c_i and c_j of A^{-1}
- Product $l = A^{-1} u$ is equal to $g_1 \times (c_i - c_j)$
- Product $r = v^T A^{-1}$ is equal to $c_i + c_j$
- $A^{-1} + \sigma A^{-1} u v^T A^{-1}$ is equal to $\sigma(A^{-1} - l r^T)$

Results of VHDL synthesis for System Generator

Program	Size	#DSP	#LUT	#FF	Latency (cycles)	# Processors	#ALPHA	#VHDL	Synt. time (seconds)
Full SM	N=3	15	750	452	$8+2N=14$	$N+1=4$	125	920	18.32
Full SM	N=7	35	1615	804	$8+2N=22$	$N+1=8$	125	920	18.64
Full SM	N=13	65	3115	1488	$8+2N=34$	$N+1=14$	125	920	18.23
Opt. SM	N=10	120	2888	2248	4	$N^2=100$	53	566	9.13
Opt. SM	N=16	288	6732	5130	4	$N^2=256$	53	566	9.64

- Full Sherman-Morrison implemented as a linear array of $N+1$ processors
- Optimized Sherman-Morrison implemented as a N^2 array in constant time
- Latency constant for Opt SM, depends linearly on N for Full SM
- Synthesis time independent of N (as expected)

Discussion

- Calculation of σ is replaced by look-up table, to avoid a division
- Numerical stability under study. Based on the selection of a subset of A^{-1} matrices among the 2^n possibilities (n being the number of switches)
- Data types in MMAlpha can be integers, fixed-points, or floats
- Other algorithms that were synthesized using MMAlpha : filters, Smith-Waterman, Kalman filters...
- Scheduling takes less than 10% of the whole synthesis time

About the Scheduler

Principle of the method

- Linear–possibly, multidimensional–schedules. $t_A(z) = \alpha_A z + \beta_A$
- For a dependence $A[z] \leftarrow B[f(z)]$, since z belongs to a polyhedral domain, make sure that $t_A(v) > t_B(f(v))$ for all vertices of the domain (plus additional constraints on rays of the domain).
- This results in the formulation of the problem as an ILP

Numbers

- Solver is the ILP program of MATHEMATICA
- Using the *interior point method*
- For the first Sherman-Morrison program, 97 variables, 442 constraints
- Solving time : 0.019221 s

Conclusion

Summary

- Expression of Sherman-Morrison Formula using a Polyhedral Equational Language
- Implementation of a fully automatic translator of ALPHA code into VHDL
- VHDL code synthesized in time independent of problem size

Future research directions

- Enlarge transformation set to target various parallel architectures
- Realize an interconnection of ALPHA and ALPHAZ (complementary tools)
- Explore algebraic transformations of ALPHA functional code

Thank you !

The two branches of the Polyhedral Model

A little bit of archeology

- Loop parallelization [Kuck, circa 1970]
- Modelization by recurrence equations [Karp et al., circa 1970]
- Systolic array modelization [Moldovan, Quinton, circa 1980]
- Data-flow analysis [Feautrier, 1991]
- Alpha language [Mauras, 1989]

Current situation

- Branch 1 : analysis of loops, dependence analysis, loop rewriting
- Branch 2 : expression of computations, program transformations
- Sharing many methods and techniques