

Efficient Computation of Algebraic Operations over Dynamically Reconfigurable Systems Specified by Rewriting-Logic Environments

M. Ayala-Rincón^{1,4}, R. B. Nogueira^{2,4}, C. Llanos^{2,4}, R. P. Jacobi^{3,4}, and R. W. Hartenstein^{5,6}

¹Departamentos de Matemática, ²Engenharia Mecânica e ³Ciência da Computação,

⁴Universidade de Brasília ayala@mat.unb.br llanos@unb.br riacobi@cic.unb.br

⁵Fachbereich Informatik, ⁶Kaiserslautern University of Technology hartenst@rhhk.uni-kl.de

Abstract

Several algebraic operations can be efficiently implement by arrays of functional units such as systolic arrays. Systolic arrays provide a large amount of parallelism. However, their applicability is restricted to a small set of computational problems due to their lack of flexibility. This limitation can be circumvented by using reconfigurable systolic arrays, where the node operations and interconnections can be redefined even at run time. In this context, several alternative systolic architectures can be explored and powerful tools are needed to model and evaluate them. Well-Known rewriting-logic environments such as ELAN and Maude can be used to specify, simulate and even synthesize complex application specific digital systems. In this paper we propose the use of rewriting-logic to model and evaluate reconfigurable systolic architectures which are applied to the efficient treatment of algebraic operations such as matrix multiplication and the FFT.

Keywords: *Term Rewriting Systems, Rewriting-Logic, Reconfigurable Systolic Arrays, Fast Fourier Transform.*

1. Introduction

The widespread popularization of mobile computing and wireless communication systems fostered the research on new architectures to efficiently deal with communications issues in hardware constrained platforms like PDAs, mobile phones and pagers, for instance. Some tasks such as data compression, encoding and decoding are better implemented through dedicated hardware modules than using standard general purpose processors (GPP). However, the exploding costs of integrated circuits fabrics associated with shorter devices lifetimes makes the design of ASIC (Application Specific Integrated Circuit) a very expensive alternative. The growing capacity of Field Programmable Gate Arrays (FPGA) and the possibility of reconfiguring them to implement different hardware architectures makes it a good solution to this rapid changing wireless market. Its flexibility opens a wide range of architectural alternatives to implement algorithms directly in hardware. In this context, it is very important to provide methods and tools to rapidly

model and evaluate different hardware architectures to implement a given algorithm.

In this paper we propose the use of rewriting-logic to specify and evaluate dynamically reconfigurable systolic hardware architectures. We show how the conceived architectures are adapted for the efficient implementation of algebraic operations such as matrix multiplication and the Fast Fourier Transform - FFT.

After the seminal work of Knuth-Bendix about the completion of algebraic equational specifications [17], rewriting has been successfully applied into different areas of computer science as an abstract formalism for assisting the simulation, verification and deduction of complex computational objects and processes. In the context of computer architectures, rewriting theory has been applied as a tool for reasoning about hardware design. To review only a reduced set of different approaches in this direction, we mention the work of Kapur who has used his well-known *Rewriting Rule Laboratory - RRL* for verifying arithmetic circuits [15,14,16] as well as Arvind's group that treated the specification of processors over simple architectures [2,24,25], the rewrite-based description and synthesis of simple logical digital circuits [27] and the description of cache protocols over memory systems [26]. Also we have contributed in this field by showing how rewriting theory can be applied for the specification of processors over simple architectures (as Arvind's group does) as well as for the purely rewrite based simulation, verification and analysis of the specified processors [3]. To achieve this we applied rewriting-logic that extends the pure rewriting paradigm allowing for a logical control of the application of the rewriting rules by strategies [21,7]. Important programming environments based on the rewriting-logic paradigm are ELAN [9,7], Maude [21,8] and Cafe-OBJ [12]. The impact of rewriting-logic as a successful programming paradigm in computer science as well as of the applicability of the related programming environments is witnessed by [20]. All our experiments were implemented in ELAN because of its great flexibility and easy manipulation of strategies. This can be done also in Maude with a bit of

additional effort. However, for effects of model checking, which can be useful for verification, Maude has been proved to be more adequate.

Section 2 provides basic concepts. Section 3 presents the rewrite based specification and simulation of systolic arrays used for simple algebraic operations such as vector and matrix multiplication. Section 4 discusses the use of rewriting-logic for specifying a dynamically reconfigurable system and efficiently implementing the FFT and section 5 is the conclusion.

2. Background

We include the minimal needed notions on rewriting and, specifically, on rewriting-logic and systolic arrays. For detailed presentations see [5] and [18,19], respectively.

2.1. Rewriting theory

A Term Rewriting System, TRS for short, is defined as a triple $\langle R, S, S_0 \rangle$, where S and R are respectively sets of *terms* and of *rewrite rules* of the form $l \rightarrow r$ if $p(l)$ being l and r terms and p a predicate and where S_0 is the subset of *initial terms* of S . l and r are called the left-hand and right-hand sides of the rule and p its condition.

In the architectural context of [25], terms and rules represent states and state transitions, respectively.

A term s can be *rewritten* or *reduced* to the term t , denoted by $s \rightarrow t$, whenever there exists a subterm s' of s that can be *transformed* according to some rewrite rule into the term s'' such that replacing the occurrence of s' in s with s'' gives t . A term that cannot be rewritten is said to be in *normal form*. The relation over S given by the previous rewrite mechanism is called the *rewrite relation* of R and is denoted by \rightarrow . Its inverse is denoted by \leftarrow and its reflexive-transitive closure by \rightarrow^* and its equivalence closure by \leftrightarrow^* .

The important notions of *terminating* and *confluence* properties are defined as usual. These notions correspond to the practical computational aspects as the determinism of processes and their finiteness.

- a TRS is said to be *terminating* if there are no infinite sequences of the form $s_0 \rightarrow s_1 \rightarrow \dots$
- a TRS is said to be *confluent* if for all *divergence* of the form $s \rightarrow^* t_1, s \rightarrow^* t_2$ there exists a term u such that $t_1 \rightarrow^* u$ and $t_2 \rightarrow^* u$.

The use of the subset of initial terms S_0 , representing possible initial states in the architectural context (which is not standard in rewriting theory), is simply to define what is a "legal" state according to the set of rewrite

rules R ; i.e., t is a legal term (or state) whenever there exists an initial state $s \in S_0$ such that $s \rightarrow^* t$.

Using these notions of rewriting one can model the operational semantics of algebraic operators and functions. Although in the pure rewriting context rules are applied in a truly non deterministic manner, in the practice it is necessary to have the control of the ordering in which rules are applied. Thus, rewriting jointly with logic, that is known as rewriting-logic, has been showed of practical applicability in this context of specification of processors since they may be adapted for discriminately representing in the necessary detail many hardware elements involved in processors.

2.2. Systolic arrays and reconfigurable systems

A systolic array is a mesh-connected pipe network of DPUs (datapath units), using only nearest neighbor (NN) interconnect. DPU functional units operate synchronously, processing streams of data that traverse the network. Systolic arrays provide a large amount of parallelism and are well adapted to a restrict set of computational problems, i.e., those which can be efficiently mapped to a regular network of operators.

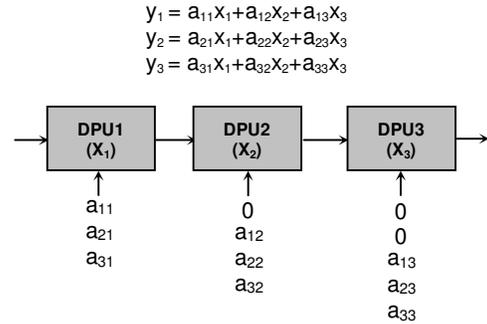


Figure 1: Vector – matrix computation

Figure 1 shows a simple systolic example of a matrix-vector multiplication. The vector elements are stored in the cells and are multiplied by the matrix elements that are shifted bottom-up. On the first cycle, the first cell (DPU1) computes $x_1 * a_{11}$, while the second and third cells (DPU2 and DPU3) multiply their values by 0. On the second cycle, the first cell computes $x_1 * a_{21}$, while the second cell computes $x_1 * a_{11} + x_2 * a_{12}$, where the first term is taken from the first cell and added to the product produced in second cell. In the third cycle, the third cell produces the first result: $y_1 = x_1 * a_{11} + x_2 * a_{12} + x_3 * a_{13}$. In the following two cycles y_2 and y_3 will be output by the third cell. Thus, by the end of the third cycle the first result is produced and the remaining values are produced in the following cycles.

There are several alternative configurations of functional cells, each one tailored to a particular class of computing problems. However, one of the main critics to systolic arrays is its restriction to applications with strictly regular data dependencies, as well as its lack of flexibility. Once designed, it is suitable to support only one particular application problem.

The limitations of systolic arrays may be circumvented by using reconfigurable circuits, the most representative of them being the FPGAs (Field Programmable Gate Arrays). Figure 2 shows the internal structure of a RAM-based FPGA. The small boxes represent the logic cells and the larger blocks, with the letter *S*, are programmable switch boxes. An FPGA can have its behavior redefined in such a way that it can implement completely different digital systems on the same chip. *Fine grain* FPGAs allow the user to define a circuit at gate level, working with bit wide operators. This kind of architecture provides a lot of flexibility, but takes more time to reconfigure than *coarse grain* reconfigurable platforms (rDPAs: reconfigurable data path arrays: arrays of rDPUs). In those ones, the user does not provide details at gate level but specify the configuration in terms of word wide operations, i.e., a functional unit is configured to operate over *n*-bit data, and the configuration just specify one among a set of available operations. The amount of configuration bits in this case is much less than in the fine grain FPGAs.

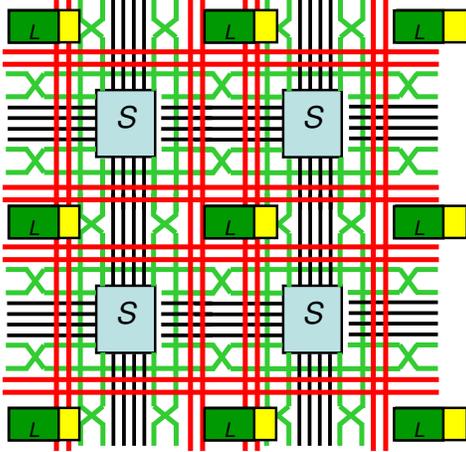


Figure 2: FPGA architecture

The design of reconfigurable systolic architectures [13, 23] aims to overcome the restriction of pure systolic circuits while keeping the benefits of a large degree of parallelism. In this approach, the operations performed by each functional unit as well as their interconnections may be reconfigured in order to be adapted to different applications. Moreover, it is possible to change the configuration of the circuit during run time, what is

called *dynamic reconfiguration*, which broadens even more the architectural alternatives. A dynamically reconfigurable system, in a given instant of time *t*, process data *d(t)* using a configuration *cfg(t)*. Instead of referring to an instruction stream and a data stream, according to Flynn taxonomy, one could describe a reconfigurable system by its *configuration stream* and its data stream. Optimization of such systems relies on a choice of a reconfigurable hardware structure and a corresponding reconfiguration scheme for a given application under a set of constraints. It is a complex task, since there are no commercial tools available that are well adapted to this kind of problem. Prototyping alternatives in VHDL or even SystemC, in a first approach, may be too cumbersome.

The variety of implementations that arise from the combination of systolic architectures and dynamically reconfigurable computing requires adequate tools for modeling and simulation of design decisions, providing a framework for design space exploration.

3. Systolic Arrays via Rewriting-Logic

Rewriting-logic based specifications of simple systolic arrays for vector and matrix multiplication are presented. In these systems each component - DPU as in the Figure 1- is called a MAC (Multiplier/Adder).

Initially, we will explain the modeling of the matrix/vector multiplier, presented in the Figure 1. The type definition for each MAC is shown in Table 1 and the structure of the systolic array in the Figure 3.

Table 1: MAC types in ELAN	
operators	global
@	: (int) Const;
port(@)	: (int) Port;
reg(@)	: (int) Reg;
[@@@@@]	: (int Port Port Reg Reg Const) MAC;
<@@@@>	: (MAC MAC MAC DataStream) Proc;
(@@@)	: (list[Data] list[Data] list[Data]) DataStream;
@	: (int) Data;
end	

Type definition in ELAN has the following syntax (Table 1): the keyword *operators* indicates the start of the type definitions, which may be *global* or *local*. Each definition is written as a rule using “:” as a separator. Its left side contains the lexical structure of a operator where the ‘@’ sign is a place holder. In the right side of the rule the types associated to the place holders as well as the name of the type are given. For instance, *Port* type is defined by `port(@)`, where the parameter between parenthesis is an integer. Each *MAC* consists of six elements: the identifier, of type *int*; two *Ports*; two *Regs* and one *Const* for the respective constant

component of the multiplier vector. The systolic processor consists of four components: three MAC's and one DataStream. The DataStream is an object with three components of type `list[Data]`.

Table 2: ELAN Description of the Sole Rule

```

rules for Proc
d1,d2,d3 : int; // input data variables
l1,l2,l3 : list[Data]; // input data list
p11,p21,p22,p31,p32:int; //ports
r11,r12,r21,r22,r31,r32:int; //regs
c1,c2,c3 : int; // constants
global
[sole]
<[1,port (p11),port (0),reg (r11),reg (r12),c1]
 [2,port (p21),port (p22),reg (r21),reg (r22),c2]
 [3,port (p31),port (p32),reg (r31),reg (r32),c3]
 (d1.l1 d2.l2 d3.l3) >
=>
  <[1,port (d1),port (0),
    reg (p11*c1),reg (0+r11),c1]
   [2, port (d2),port (r12),reg (p21*c2),
    reg (p22+r21),c2]
   [3, port (d3),port (r22),reg (p31*c3),
    reg (p32+r31),c3]
  (l1 l2 l3) >
end
end

```

The rule `sole` given in the Table 2 describes the behavior of the processor during one cycle of the execution: after one-step of reduction, applying this rule, all necessary changes in the specified processor are done. Firstly, notice that `d1`, `d2` and `d3` at the top of the DataStream, are removed from the three lists of data and placed into the first ports of the three MACs.

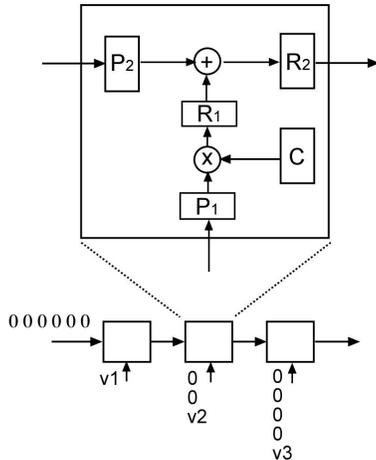


Figure 3: MAC Systolic Array Architecture.

Afterwards, notice that the multiplications between the contents of each first port `pi1` and the corresponding constant `ci` are placed in the first register of each MAC, for $i=1,2$ and 3 and the additions between the first register `ri1` and the second port `pi2` are placed in the second port of each MAC, for $i=1,2$ and 3 .

Finally, note that the transfer of data from the second register `ri2` of each MAC to the second port of the next component `p(i+1)2` is done, for $i=1, 2$. This is done by only one application of the rewriting rule `sole`

simultaneously. Notice the necessity of the extra zeros with respect to the original proposal in the Figure 1.

A simple mechanism of reconfiguration is the possibility of changing the constants in each MAC. Then a computation with our systolic array consists of two phases: a reconfiguration phase, where the constants are set and the subsequent processor execution phase with the previously defined rule `sole`.

Table 3: conf Rule for Reconfiguration

```

[conf]
<[1,port (p11),port (0),reg (r11),reg (r12),c1]
 [2,port (p21),port (p22),
    reg (r21),reg (r22),c2]
 [3,port (p31),port (p32),
    reg (r31),reg (r32),c3]
 (d1.l1 d2.l2 d3.l3) >
=>
  <[1,port (p11),port (0),reg (r11),reg (r12),1]
   [2,port (p21),port (p22),reg (r21),reg (r22),0]
   [3,port (p31),port (p32),reg (r31),reg (r32),0]
  (d1.l1 d2.l2 d3.l3) >
end
strategies for Proc
implicit
[] withconf => conf; normalise(sole) end
[] simple => normalise(sole) end
end

```

The Table 3 shows one additional rule created for the reconfiguration of a processor called `conf`. It simply changes the contents of the constant part of each MAC (by the vector $(1,0,0)$). Note that with the pure rewriting based paradigm this rule applies infinitely. Thus for controlling its application, we define a logical strategy, called `withconf`, which allows for the execution of one-step of reduction with the rule `conf` (the first reconfiguration stage) and a normalization with the rule `sole` (the second processor execution stage).

The Figure 4 shows the structure of a systolic array for 4×4 matrix multiplication. Its description is given in the Table 4. The approach adopted here is different from the previous one in order to reduce the number of variables needed for its description. One solution is to split the cycle defining independent rewriting rules, to be applied under a reasonable strategy, to simulate the internal process into each MAC component and the propagation of data between each component to their North and East connected MACs.

We define a rule for each of the sixteen components, which propagates the contents into their registers two and three to their North and East connected components, respectively.

To complete a whole cycle of execution, as consequence of the direction in which data is transferred between the MACs, these sixteen rules should be applied right-left and top-down.

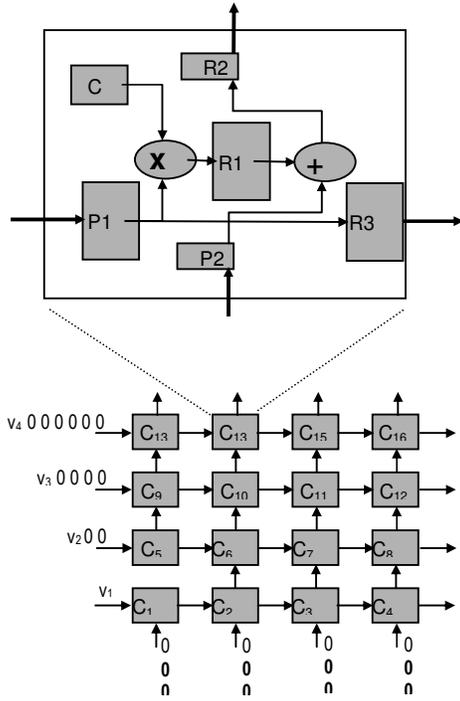


Figure 4: Systolic for matrix multiplication

Table 4: a 4x4 Systolic array Description

```

operators global
@      : ( int ) Const;
p(@)   : ( int ) Port;
r(@)   : ( int ) Reg;
[ @, @, @, @, @, @, @ ] : ( int Port Port Reg Reg Reg
Const) MAC;
< @
  @ @ @ @
  @ @ @ @
  @ @ @ @
  @ @ @ @
  @ >
: ( DataString
  MAC MAC MAC MAC // MACs 13 14 15 16
  MAC MAC MAC MAC // MACs 09 10 11 11
  MAC MAC MAC MAC // MACs 05 06 07 08
  MAC MAC MAC MAC // MACs 01 02 03 04
  DataString ) Proc;
( @@@@ ) : ( list[Data] list[Data]
  list[Data] list[Data] ) DataString;
@      : ( int ) Data;
end

```

All these rules are very similar and one of them is presented in the Table 5. Observe that the rules for the South (mac01, mac02, mac03, mac04) and West (mac01, mac05, mac09, mac13) boundary components of the processor load the data (dS and dW) from the head of the corresponding list of the data stream (lS1, lS2, lS3, lS4 and lW1, lW2, lW3 and lW4). Also observe that the rules for MACs in the North (mac13, mac14, mac15, mac16) and East (mac04, mac08, mac12, mac16) boundaries of the processor only transfer data to the East and North neighbor MACs, respectively; except, of course, for mac16. Thus, to complete a cycle of the processor, different orderings of application of these rules are possible. In the Table 6 we present a possible strategy called onecycle which defines an(other) ordering of application for completing

a cycle of the processor. For completing the simulation of execution with this simple processor, one should define a normalization based on this strategy: normalise(onecycle). The built-in strategy normalise applies onecycle until a normal form is reached.

Table 5: a set of rules for matrix-vector multiplier

```

rules for Proc
m01,m02,m03,m04,m05,m06,m07,m08: MAC; // 1-8 MACs
m09,m10,m11,m12,m13,m14,m15,m16:MAC; //9-16 MACs
dW, dS      : int; // data East and South
IW1,IW2,IW3,IW4,IS1,IS2,IS3,IS4:list[Data]; // West and South
r1,r2, r3,rN1,rN2,rN3 : int; // Central North and
rE1,rE2,rE3      : int; // East registers 1,2,3
p1,p2,pN1,pN2,pE1,pE2: int; //Central,North and East ports
c,cE,cN          : int;
global
[mac16]
< (IW1 IW2 IW3 IW4)
  m13 m14 m15 [16,p(p1),p(p2),r(r1),r(r2),r(r3),c ]
  m09 m10 m11 m12
  m05 m06 m07 m08
  m01 m02 m03 m04
  (IS1 IS2 IS3 IS4) >      =>
< (IW1 IW2 IW3 IW4)
  m13 m14 m15 [16,p(p1),p(p2),r(p1*c),r(r1+p2),r(p1),c ]
  m09 m10 m11 m12
  m05 m06 m07 m08
  m01 m02 m03 m04
  (IS1 IS2 IS3 IS4) >
end
...
end

```

In this rewriting-logic setting our specification could be easily modified to allow the interpretation of parts of the processors as reconfigurable components. At first glance, one could look at the constants of the 16 MACs as a reconfigurable component. In this way the processor can be adapted to be either a 4-vector versus 4x4-matrix multiplier or vice-versa and the 4x4-matrix may be modified to represent, for example, either the identity or the F_4 matrix of the Discrete Fourier Transform - DFT, which is discussed in next section.

Table 6: onecycle strategy for rule application

```

Strategies for Proc
implicit
[ ] onecycle =>
  mac16;mac15;mac14;mac13;
  mac12;mac11;mac10;mac09;
  mac08;mac07;mac06;mac05;
  mac04;mac03;mac02;mac01
end
end

```

4. Run time efficient FFT Modeling

The FFT is an implementation of the DFT, which is widely used in signal processing. Given an n-array of complex numbers $a = (a_0, \dots, a_{n-1})$, its DFT, $F_n \times a$, is the n-array (b_0, \dots, b_{n-1}) , where

$$b_j = \sum_{k=0}^{n-1} a_k \cdot \omega_n^{kj} \text{ for } j=0,1,\dots,n-1 \quad \text{and}$$

$\omega_n = e^{i \frac{2\pi}{n}}$ is a primitive n^{th} complex root of the unity.

The basic operations are multiply-accumulate, executed over complex numbers.

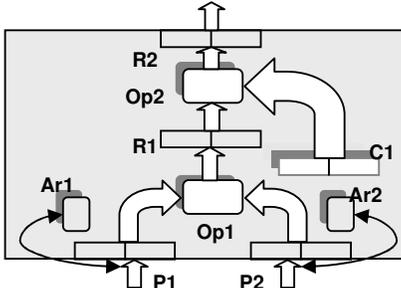


Figure 5: node architecture for FFT

The FFT is an $O(n \ln n)$ run time implementation of DFT based on a recursive algorithm proposed by Cooley-Tukey. This algorithm can be implemented in dataflow hardware as presented in classical text books on algorithms [10,6,1]. The number of data points is a power of 2. The network of nodes is a butterfly circuit. Each node implements a complex number multiplies-accumulate operation on its inputs: $b_j = u_j + z v_j$.

The *two 8-array* architecture that we use for computing F_8 is based on these circuits and its (operational semantics and) correctness is founded on the adequate application of dynamic reconfiguration of the operators, constants and data selection registers. Reconfiguration and execution steps run simultaneously alternated on the two 8-array of MACs. The structure of each MAC is presented in the Figure 5. We distinguish between reconfigurable (shadowed) and fixed components. The formers are: data selection registers, Ar1 and Ar2; operators, Op1 and Op2; and constant, C1. The latter are the ports and registers: P1, P2 and R1 and R2.

The registers, ports and the constant store complex numbers and consist of two components: real and imaginary. The operators can be reconfigured as any operation over complex numbers. In particular, for implementing FFT we will use only addition (+), subtraction (-) and multiplication (\times). The two data selection registers, Ar1 and Ar2, are used to indicate in each of the eight MACs of one of the two 8-arrays the origin of the data that should be loaded into the respective ports, P1 and P2. The options for configuration of these address registers are either the input (I) (as input we will supply the coefficients of a given polynomial permuted adequately) or the output (second register R2) of one of the eight nodes of the opposite 8-array of MACs (indexed by 0,1,...,7). In any reconfiguration the constant is set with arbitrary complex numbers. For computing FFT, we will set these constants with the adequate complex roots of the unity.

4.1 The two 8-array of MACs system

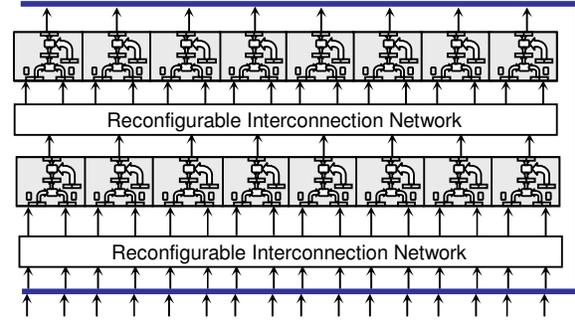


Figure 6. Two 8-array system

The Figure 6 shows the basic idea behind the two 8-array system. The *North* and *South* rows are composed by 8 nodes with the architecture depicted in the Figure 5. The node outputs of a row are feedback to the inputs of the other row through a reconfigurable interconnection network (RIN). The RIN can provide to the MAC ports any MAC output or an external input. The configuration of data selection registers Ar1 and Ar2 will select from the RIN the specific node inputs in a given iteration of the algorithm. In the first step, one of the 8-array receives as input zeros and coefficients of an input polynomial $a_0 + a_1 x + \dots + a_7 x^7$ in the adequate ordering (bit-reversal permutation), taken from the primary (external) inputs. Then, at each step the interconnections and the node operations are reconfigured in order to implement the corresponding butterfly slice alternating from a row to the other. In this way while the MACs in one row are executing the others are being reconfigured, which eliminates from the run time analysis the time spent for reconfiguration except for the time spent for the initial reconfiguration. The initial reconfiguration parameters are given by the sequence:

- 0 0:** I,I,+1, \times ; **1:** I,I,+1, \times ; **2:** I,I,+1, \times ; **3:** I,I,+1, \times ;
4: I,I,+1, \times ; **5:** I,I,+1, \times ; **6:** I,I,+1, \times ; **7:** I,I,+1, \times ;

The first zero stands for indicating that the North row is being reconfigured while the South row is executing vacuous operations. The other parameters of reconfiguration indicate that the node 0 receives its inputs from the corresponding external inputs; its first operator is configured as addition; its constant component as 1; and its second operator as multiplication. Similarly for the remaining seven nodes. After this reconfiguration, the operations in the north row are executed while the system is being reconfigured according to the parameters:

- 1 0:** 0,1,+1, \times ; **1:** 0,1,-1, \times ;
2: 2,3,+1, \times ; **3:** 2,3,-, i, \times ;

Table 7: Description of the Operators of the Two 8-array Architecture

operators global	
+ : Op; - : Op; * : Op;	
< @ >	: (Op) OpUnit;
< @ @ >	: (num num) complexUnit;
const(@)	: (complexUnit) Const;
port(@)	: (complexUnit) Port;
reg(@)	: (complexUnit) Reg;
addr(@)	: (int) Addr;
@, @, @, @, @	: (int Port Port Reg Reg) fixMAC;
@, @, @, @, @	: (Addr Addr Const OpUnit OpUnit) recMAC;
[@ # @]	: (fixMAC recMAC) MAC;
MACsArray(@ @ @ @ @ @ @ @)	: (MAC MAC MAC MAC MAC MAC MAC MAC) MACsArray;
reconfigure(@, @)	: (MACsArray ReconfParameter) MACsArray;
propagateRegsValuesFromTo(@, @)	: (MACsArray MACsArray) MACsArray;
operate(@, @, @)	: (complexUnit complexUnit OpUnit) complexUnit;
getRecMAC(@)	: (MACConfig) recMAC;
getMACInit(@, @, @)	: (int complexUnit complexUnit) MAC;
getMAC(@, @)	: (MAC MACsArray) MAC;
extractRegValue(@)	: (MAC) complexUnit;
(@ @ @ @ @ @ @ @)	: (int int num num Op Op) MACConfig;
< @ @ @ @ @ @ @ @ >	: (MACConfig MACConfig MACConfig MACConfig MACConfig MACConfig MACConfig MACConfig)ReconfParameter;
continue	: ReconfParameter; // vacuous reconfiguration
< @ @ @ @ >	: (int list[ReconfParameter] MACsArray MACsArray) Proc;
end	

4: 4,5,+1, ×; **5:** 4,5,-1, ×;
6: 6,7,+1, ×; **7:** 6,7,-, i, ×;

2: 2,6,+1, ×; **3:** 3,7,+1, ×;
4: 0,4,-1, ×; **5:** 1,5,-1, ×;
6: 2,6,-,1, ×; **7:** 3,7,-,1, ×;

Execution in the North row gives in the output register (R_2) of each node the coefficients: $a_0, a_4, a_2, a_6, a_1, a_5, a_3$ and a_7 , respectively. Observe that this second step provides again the same input, but now, adjusted to be processed in the South row that is being simultaneously reconfigured according to the above parameters. The first "1" in the above reconfiguration parameters means that the South row is being reconfigured while the North row is executing as it has been explained. The other reconfiguration parameters mean that the first and second data selection registers of the nodes 0 and 1 should be loaded with 0 and 1. Thus, the outputs of nodes 0 and 1 are loaded in the associated ports, and these are added in the first node and subtracted in the second node. All nodes are configured with the constant 1 in this iteration except for the fourth and eighth where the constant is the complex i . The second operator remains as multiplication. After this second reconfiguration and the third execution over the South row (while the North row is being reconfigured) we will obtain as respective outputs the values: $a_0+a_4, a_0-a_4, a_2+i-a_6, a_2-i-a_6, a_1+a_3, a_1-a_3, a_5+i-a_7$ and a_5-i-a_7 .

The third reconfiguration is given by the sequence:

0 0: 0, 2, +,1, ×; **1:** 1,3,+, 1, ×;
2: 0, 2, -, 1, ×; **3:** 1,3, -, 1, ×;
4: 4, 6, +,1, ×; **5:** 5,7,+, (1+i)/, ×;
6: 4, 6, -, i, ×; **7:** 5,7, -, (-1+i)/, ×;

Finally, simultaneously to the fourth execution phase, the 8-array is reconfigured with the following sequence:

1 0: 0,4,+1, ×; **1:** 1,5,+1, ×;

This gives as output $F_8 \times (a_0, \dots, a_7)$, that is the DFT of the polynomial $a_0 + a_1 \cdot x + \dots + a_7 \cdot x^7$.

4.2 Specification of the two 8-array in ELAN

The key operators of our ELAN specification of this system have the type description given in the Table 7. Notation "<@ @> : (num num) complexUnit;" means that "< >" is a binary operator of type complexUnit with two parameters of type num.

Our system is described as the operator:

```
< @ @ @ @ > : ( int list[ReconfParameter]
                MACsArray MACsArray ) Proc;
```

whose last two parameters are the two 8-arrays of MACs of type **MACsArray**, the first parameter of type int identifies the 8-array being reconfigured and the second parameter is a list of reconfiguration parameters. Each **MACsArray** consists of eight MACs being the operator MAC defined by "[@ # @] : (**fixMAC** **recMAC**)", where **fixMAC** and **recMAC** are the types of the operators for its fixed and reconfigurable parts, as described in the Figure 5.

Each simultaneous execution-reconfiguration step of this system is specified by rewriting rules as the one presented in the Table 8. This rule changes the first (North) 8-array **MACsArray1** to **MACsArray1Res** by applying the **EXECUTE** strategy:

```
MACsArray1Res :=(EXECUTE) MACsArray1
```

while the second (South) 8-array **MACsArray2** is being reconfigured according to the head parameter of reconfiguration **recfpar** in the reconfiguration stream **recfpar.streamrecf**:

```
MACsArrayAux := () reconfigure (MACsArray2, recfpar)
```

Table 8: Rule of Execution-Reconfiguration

```
[oneCycle] // Execution-reconfiguration in
// the first and secd 8-array, resp.
< 0 recfpar.streamrecf MACsArray1 MACsArray2 >
=>
  < 1 streamrecf MACsArray1Res MACsArray2Res >
  where MACsArray1Res := (EXECUTE) MACsArray1
  where MACsArrayAux := ()
        reconfigure ( MACsArray2, recfpar )
  where MACsArray2Res := ()
        propagateRegsValuesFromTo (MACsArray1Res,
                                   MACsArrayAux )
End
```

The second 8-array finishes this step loading their ports according to the address selection registers of its MACs with the corresponding output registers of the first 8-array. The last is done by means of the operator **propagateRegsValuesFromTo**. All operators are defined by rewriting rules.

Table 9: Rule of Execution in the MACs

```
[MAC01] // Execution over the first and second
// MACs (MAC0 and MAC1) of one 8-array
MACsArray (
  [0, port (cPort1), port (cPort2),
   reg (cReg1), reg (cReg2) #
   addr1, addr2, const (cConst1), op1, op2
  ]
  [1, port (cPort3), port (cPort4),
   reg (cReg3), reg (cReg4) #
   addr3, addr4, const (cConst2), op3, op4
  ]
  [ fix2#rec2 ] [ fix3#rec3 ] [ fix4#rec4 ]
  [ fix5#rec5 ] [ fix6#rec6 ] [ fix7#rec7 ] )
=>
  MACsArray (
    [0, port (cPort1), port (cPort2),
     reg (cRegRes1), reg (cRegRes2) #
     addr1, addr2, const (cConst1), op1, op2 ]
    [1, port (cPort3), port (cPort4),
     reg (cRegRes3), reg (cRegRes4) #
     addr3, addr4, const (cConst2), op3, op4 ]
    [ fix2#rec2 ] [ fix3#rec3 ] [ fix4#rec4 ]
    [ fix5#rec5 ] [ fix6#rec6 ] [ fix7#rec7 ] )
  where cRegRes1 := () operate (cPort1, cPort2, op1)
  where cRegRes2 := ()
        operate (cRegRes1, cConst1, op2)
  where cRegRes3 := () operate (cPort3, cPort4, op3)
  where cRegRes4 := ()
        operate (cRegRes3, cConst2, op4 )
end
```

The execution cycle is split in four rewriting rules (MAC01, MAC23, MAC45, MAC67) for pairs of MACs. The specification of the rule MAC01 for the first pairs of MACs of one 8-array is presented in the Table 9. In this rule the values in the ports of the first two MACs are operated according to the configuration of the first operator in each MAC (**cRegRes1 := () operate (cPort1, cPort2, op1)** and **cRegRes3 := () operate (cPort3, cPort4, op3)**); then these results, which are loaded in the first register of the corresponding MACs, are operated, according to the configuration of the second operator, with the

configured constants (**cRegRes2 := () operate (cRegRes1, cConst1, op2)** and **cRegRes4 := () operate (cRegRes3, cConst2, op4)**) and the results are loaded in the second register of each MAC.

The execution over an 8-array of MACs is implemented via the logical strategy EXECUTE => MAC01; MAC23; MAC45; MAC07. In fact, in theory a unique rule is necessary for the execution, but it is done in this way because of a restriction in ELAN in the maximum number of different variables that one can use in the description of a rewriting rule.

Table 10: Rule of Dynamical Reconfiguration

```
[] reconfigure (MACsArray (
  [ fix0 # rec0 ] [ fix1 # rec1 ]
  [ fix2 # rec2 ] [ fix3 # rec3 ]
  [ fix4 # rec4 ] [ fix5 # rec5 ]
  [ fix6 # rec6 ] [ fix7 # rec7 ] ),
  < MACConfig0 MACConfig1 MACConfig2 MACConfig3
    MACConfig4 MACConfig5 MACConfig6 MACConfig7 >
=>
  MACsArray ([ fix0 # getRecMAC (MACConfig0) ]
             [ fix1 # getRecMAC (MACConfig1) ]
             [ fix2 # getRecMAC (MACConfig2) ]
             [ fix3 # getRecMAC (MACConfig3) ]
             [ fix4 # getRecMAC (MACConfig4) ]
             [ fix5 # getRecMAC (MACConfig5) ]
             [ fix6 # getRecMAC (MACConfig6) ]
             [ fix7 # getRecMAC (MACConfig7) ] )
end
```

The reconfiguration over an 8-array (which is applied simultaneously to the previously described execution over the other 8-array) is guided by the rewriting rule in the Table 10. The first argument of the operator reconfigure is an 8-array of MACs whose MACs are reconfigured according to the reconfiguration parameters given by eight arguments of type MACConfig (see the Table 7). Each of these arguments include two values for the address selection registers, two numbers for the reconfigurable constant (real and complex part) and two values for the reconfiguration of the operations.

As input of this system both data and a reconfiguration stream are given. When no reconfiguration is necessary one can use a reconfiguration called **continue** with vacuous effect over the reconfigurable part of each MAC.

Now we explain how we use logical strategies for simulating the desired execution with the simultaneous dynamic reconfigurations. The key for a correct simulation of our processor is in fact a very simple logical strategy, which simulates the execution-reconfiguration steps. The former corresponds to the use of the strategy EXECUTE and the latter to the execution of the rewriting rules of reconfiguration (see the Table 8). The logical strategy PROCESS for controlling the execution-reconfiguration of the process is specified as:

```

strategies for Proc
implicit
  [] PROCESS => input; repeat*(oneCycle);
                output
end
end
end

```

PROCESS basically organizes the application of rules for propagating the input data and reconfiguration stream, repeating the The `oneCycle` rules (see the Table 8) as long as possible and then giving the output (i.e., the contents of the register 2 of the MACs belonging to the 8-array in execution during the last cycle). The use of logical strategies for guiding the application of rules in ELAN allows for a natural separation between the execution and reconfiguration steps in our proposed processors. We believe that this is a *clean* way to specify and simulate this kind of (dynamically) reconfigurable architectures. By *clean* we mean in a realistically manner in relation to eventual physical implementations of the conceived hardware.

By providing appropriate reconfiguration streams this two 8-array system can be adapted to solve other operations, like matrix multiplication, inverse of the DFT, string matching, etc.

It should be stressed here that one of the main advantages of this rewriting formalism is the direct reduction of the correctness proof of our specification of the FFT to the usual algebraic proof as presented in [6].

3.3 A physical in-place implementation of FFT

Our system has used two 8-arrays in order to alternate execution-reconfiguration steps which are alternatively executed simultaneously during each cycle. In this way time for reconfiguration is discarded from the run time complexity. This makes as efficient our implementation of the FFT as the usual software implementations. This is possible since computing of operations with complex numbers takes longer time than reconfiguration time eliminating the reconfiguration overhead. But our system is not space optimal for implementing the FFT. In fact, in a system consisting of a sole 8-array of MACs, steps of reconfiguration and execution can be alternated. In this approach, the data processing must be interrupted while reconfiguration takes place. And over this one 8-array system it is possible to implement the FFT alternating reconfigurations and steps of the computation of the FFT. The use of a unique array of MACs makes this proposed physical system optimal in the use of space such as the well-known *in place* algorithmic solutions of the FFT [4]. Of course, in this one 8-array system we have to take in count, for computing the run time complexity, the time required

for reconfiguration. For both proposed systems, the number of necessary reconfigurations and execution steps for computing F_8 is four (and in the general case $\ln(n)+1$).

The one 8-array architecture was modeled and simulated in ELAN, using a similar approach. Descriptions of the implementations are not presented here due to space limitations, but they are available in our web site: www.mat.unb.br/~ayala/TCgroup.

Although our specifications were proved correct, we have verified their correct functionality, even for complex polynomials, by comparing our outputs with the ones given by the algebraic system Maple.

5. Conclusions

The examples in the paper describe reconfiguration using rewriting-logic strategies. Representing the reconfiguration in this way, outside of the rewrite rules, seem unnecessary: one can argue that this can be expressed as rules using conditions on appropriate state variables - functional approaches for describing digital circuits is nothing new [11] -. But in our rewriting-logic based setting, we showed how one can naturally profit from the discrimination between rewriting and logical strategies to simplify the purely rewrite based specification, experimentation, simulation (and even verification [3]) of reconfigurable systems. By rewriting-logic even the sophisticated dynamical reconfiguration appears a very natural mechanism to be simulated via logical strategies.

Since digital systems get more and more complex, modeling the various architectural trade offs in the context of reconfigurable systems may benefit from the high abstraction level provided by rewriting-logic environments. Our experiments with ELAN targeted reconfigurable systolic arrays and their use for the efficient implementation of algebraic operators. For the implementation of complex operators such as the FFT, we have conceived physical systems, which are run time efficient ($O(n \ln n)$) as well as space efficient (*in place*).

Hardware description languages like VHDL, Verilog, and SystemC, do not provide the degree of abstraction and flexibility found in rewriting(-logic) systems. In fact, they do not compete in this field, since the detailed hardware design still must pass through a hardware description language (VHDL is the “assembly language” in this context). We do not need their architectural and circuit details for mapping an application onto a rDPA, nor design space exploration to optimize, for instance, KressArray platforms [22].

Currently, to study the possibilities of dynamical reconfiguration more sophisticated models are under development. Additionally, as future work we propose the automatic generation of synthesizable VHDL models from the ELAN descriptions.

6. References

- [1] S. G. Akl. *Parallel Computation: Models and Methods*. Prentice-Hall, 1997.
- [2] Arvind and X. Shen, *Using Term Rewriting Systems to Design and Verify Processors*, Technical Report 419, Laboratory for Computer Science - MIT, 1999. Also in IEEE Micro Special Issue on Modeling and Validation of Microprocessors, 1999.
- [3] M. Ayala-Rincón, R. M. Neto, R.P. Jacobi, C. H. Llanos and R. W. Hartenstein, *Applying ELAN Strategies in Simulating Processors over Simple Architectures*. In B. Gramlich and S. Lucas Eds., *Reduction Strategies in Rewriting and Programming*, ENTCS 70(6):20 pages, 2002.
- [4] M. Ayala-Rincón, R. B. Nogueira, R.P. Jacobi, C. H. Llanos and R. W. Hartenstein, *Modeling a Reconfigurable System for Computing the FFT in Place via Rewriting-Logic*. To appear in Proc. SBCCI 2003.
- [5] F. Baader and T. Nipkow, *Term Rewriting and all That*, Cambridge University Press, 1998.
- [6] S. Baase and A. van Gelder, *Computer Algorithms: Introduction to Design and Analysis*, Addison-Wesley, 1999.
- [7] P. Borovanský, C. Kirchner, H. Kirchner and P.-E. Moreau, *ELAN from a rewriting logic point of view*, pages 155-185 of [20].
- [8] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and J. F. Quesada, *Maude: specification and programming in rewriting logic*, pages 187-243 of [20].
- [9] H. Cirstea and C. Kirchner, *Combining Higher-Order and First-Order Computation Using rho-Calculus: Towards a Semantics of ELAN*, Chapter 6 in D. M. Gabbay and M. de Rijke, Eds., *Frontiers of Combining Systems 2*, Studies on Logic and Computation, 7, pages 95-121, Research Studies Press/Wiley, 1999.
- [10] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms*, The MIT Press, 2001.
- [11] K. Claessen, P. Bjesse, M. Sheeran and S. Singh, *Lava: Hardware Design in Haskell*, in Proc. ICFP, pages 174-184, 1998.
- [12] R. Diaconescu and K. Futatsugi, *Logical foundations of CafeOBJ*, pages 289-318 of [20].
- [13] R. Hartenstein, R. Kress and H. Reinig, *A Scalable, Parallel and Reconfigurable Datapath Architecture*. Sixth International Symposium on IC Technology, Systems and Applications - ISIC'95, Singapore, 1995. Available at www.kressarray.de.
- [14] D. Kapur. *Theorem Proving Support for Hardware Verification*, invited talk Third Int. Workshop on First-Order Theorem Proving, St. Andrews, Scotland, 2000.
- [15] D. Kapur and M. Subramaniam. *Using and Induction Prover for Verifying Arithmetic Circuits*. Journal of Software Tools for Technology Transfer. 3(1):32-65, Springer Verlag, 2000.
- [16] D. Kapur and M. Subramaniam, *Mechanizing Verification of Arithmetic Circuits: SRT Division*. In Proc. 7th Conf. on Foundations of Software Technology and Theoretical Computer Science. Vol. 1346 of LNCS, Springer, 1997.
- [17] D. E. Knuth and P. B. Bendix. *Computational Problems in Abstract Algebra*, chapter Simple Word Problems in Universal Algebras, pages 263-297. J. Leech, ed., Pergamon Press, Oxford, 1970.
- [18] H.T. Kung, C. E. Leiserson, *Systolic Arrays for VLSI; Sparse Matrix Proc.* 1978, Society for Industrial and Applied Mathematics, 1979, pages 256-282.
- [19] S. Y. Kung. *VLSI Array Processors*. Prentice-Hall, 1987.
- [20] N. Martí-Oliet and J. Meseguer, eds., *Special issue on Rewriting Logic and its Applications*, Theoretical Computer Science 285(2): 119-564, 2002.
- [21] J. Meseguer. *Rewriting Logic and Maude: Concepts and Applications*, In L. Bachmair Ed., Eleventh Int. Conf. on Rewriting Techniques and Applications RTA 2000, LNCS, Vol. 1833, pages 1-26, Springer, 2000.
- [22] U. Nageldinger. *Coarse-Grained Reconfigurable Architecture DesignSpace Exploration*. Dissertation, Univ.Kaiserslautern, June 1, 2001.
- [23] R. Hartenstein, M. Herz, T. Hoffmann, U. Nageldinger. *Kress Array Explorer: A New CAD Environment to Optimize Reconfigurable Datapath Array Architectures*. 5th Asia and South Pacific Design Automation Conference - ASP-DAC 2000, Yodohama, Japan, 2000. Available at www.kressarray.de.
- [24] X. Shen and Arvind, *Design and Verification of Speculative Processors*, Technical Report 400A, Laboratory for Computer Science - MIT, 1998. Also in Proc. of the Workshop on Formal Techniques for Hardware and Hardware-like Systems, Marstrand, Sweden, 1998.
- [25] X. Shen and Arvind, *Modeling and Verification of ISA Implementations*, Technical Report 400B, Laboratory for Computer Science - MIT, 1998. Also in Proc. of the Australasian Computer Architecture Conference, Perth, Australia, 1998.
- [26] X. Shen, Arvind and L. Rudolph, *CACHET: an adaptive cache coherence protocol for distributed shared-memory systems*, ACM International Conference on Supercomputing, pages 135-144, 1999.
- [27] J.C. Hoe and Arvind, *Hardware Synthesis from Term Rewriting Systems*, Laboratory for Computer Science - MIT, 421 A, 1999. Also in Proc. of the Tenth IFIP International Conference on VLSI - VLSI 1999.