Prototyping Time and Space Efficient Computations of Algebraic Operations over Dynamically Reconfigurable Systems Modeled by Rewriting-Logic*

M. AYALA-RINCÓN[1], AND C. H. LLANOS[2], AND R. P. JACOBI[3]
Universidade de Brasília
and
R. W. HARTENSTEIN[4]
Technische Universität Kaiserslautern

Many algebraic operations can be efficiently implemented as pipe networks in arrays of functional units such as systolic arrays that provide large amount of parallelism. However, the applicability of classical systolic arrays is restricted to problems with strictly regular data dependencies yielding only arrays with uniform linear pipes. This limitation can be circumvented by using reconfigurable systolic arrays or reconfigurable data path arrays, where the node interconnections and operations can be redefined even at run time. In this context, several alternative reconfigurable 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 and simulate complex application specific integrated systems. In this paper we propose a methodology based on rewriting-logic which is adequate to quickly model and evaluate reconfigurable architectures (RA) in general and, in particular, reconfigurable systolic architectures. As an interesting case study we apply this rewriting-logic modeling methodology to the space-efficient treatment of the FFT. The FFT prototype conceived in this way, has been specified and validated in VHDL using the Quartus II system.

Categories and Subject Descriptors: C.1 [Processor Architectures]: Other Architecture Styles; B6 [Logic Design]: Design Styles; F4 [Theory of Computation]: Grammars and Other Rewriting Systems, Algebraic Manipulation
General Terms: Design, Experimentation, Theory
Additional Key Words and Phrases: Term Rewriting Systems (TRS), rewriting-logic, reconfigurable computing, dynamically reconfigurable systems, systolic arrays, Fast Fourier Transform (FFT)

1. INTRODUCTION

Systems on Chip (Soc) often include reconfigurable architectures (RA) in addition to other modules like processors, memories, application-specific circuits, I/O units, etc. IP modules can be provided in several flavors, such as a layout description, a programmable logic device (PLD) bit stream or a synthesizable HDL model. It may be implemented as a software module, executed by a digital signal processor (DSP) or by a dedicated hardware device. But an implementation on a RA is preferred for particular application areas, like for instance, mobile communication systems which must repetitively adapt themselves to rapidly evolving standards, and changing communication parameters, like bandwidth, protocols, and so on. Moreover, the low power requirement for battery powered devices is another key factor for mobile computing [ABNOUS et al. 1998].

*This research was supported by the Brazilian and German Foundations CAPES and DFG.
[1] Corresponding author, partially funded by the Brazilian research council CNPq; Departamento de Matemática, Universidade de Brasília, 70900-010 Brasília D. F., Brasil, ayala@mat.unb.br; [2]Departamento de Engenharia Mecatrônica, llanos@unb.br; [3]Departamento de Ciência da Computação, rjacobi@cic.unb.br; [4]Fachbereich Informatik, Technische Universität Kaiserslautern, hartenst@rhrk.uni-kl.de.

Software vs. Configware. Reconfigurable Logic (RL) has become mainstream in embedded systems. Since terminology is not yet well established and most CE/CS-related undergraduate curricula still ignore RL, some fundamental explanations seem to be useful. The programming source for CPUs is software and its semantics is procedural (an instruction schedule). RL, however, is programmed by configware compiled (for placement and routing, not instruction scheduling) into reconfiguration bit code downloaded to the FPGA's hidden RAM. The semantics of configware is structural (not procedural) [BECKER AND HARTENSTEIN 2003]. In contrast to CPUs, RL does not do any instruction fetch at run time.

Reconfigurable Logic (RL) vs. Reconfigurable Computing (RC). FPGAs use thousands of configurable logic blocks (CLBs), mostly only a single bit wide - a kind of logic design issue. A relatively new alternative to RL is RC using reconfigurable datapath arrays (rPDAs) [HARTENSTEIN 2001], which are a kind of reconfigurable systolic array where - instead of CLBs - reconfigurable Data Path Units (rDPUs) are used, being multiple bits wide like 32 bits, for instance. Note, that not having a program counter, a rDPU is not a CPU. This alternative paradigm uses data counters instead, which are located in auto-sequencing data memory banks [HERZ et al. 2002]. A configware compiler for RC (e. g. [BECKER et al. 1998]) creates a tailored pipe network to be configured into the rDPA. This is not just an abstraction since the pipe network is physically implemented at RT level

Rapid Prototyping. rDPAs are available as IP cores, but not yet as COTS (Commercial off-the-shelf) in quantities. Some rDPA vendors offer a prototyping board featuring a physical rDPA (e. g. [PACTCORP]). But also a design space explorer [NAGELDINGER et al. 2000] may be used to test area efficiency, to estimate power dissipation, etc. To emulate an application (e. g., if simulation is too slow) FPGAs may be used for rapid prototyping.

Rewriting-Logic. The **main proposal** of this work is the use of rewriting-logic as a general and adequate discipline to specify, verify, and evaluate architectural systems and in particular RAs and rDPAs. Important rewriting-logic computational environments such as ELAN [CIRSTEA AND KIRCHNER 2000, BOROVANSKÝ et al. 2002], Maude [MESEGUER 2000, CLAVEL et al. 2002] and Cafe-OBJ [DIACONESCU AND FUTATSUGI 2002] have been applied to hardware design and verification [MESEGUER 2003], AYALA-RINCÓN et al. [2002, 2003, 2004]. All our experiments were implemented in ELAN because of its great flexibility and easy manipulation of strategies. Although our work is restricted to design approaches based on rewriting(-logic), it is important to mention here that other methodologies have been developed to specify (dynamically) reconfigurable systems. Some of them have been shortly referenced in [SHIRAZI et. al. 2000], also supporting run-time reconfigurable designs and involving Xilinx tools for Virtex FPGAs.

Fast Fourier Transform. RAs have become mainstream, not only in embedded systems. A popular target area is *Fast Fourier Transform* (FFT), widely used due to its variety of applications in DSP [VERGARA et al. 1998]. As a non trivial case study of the proposed rewriting-logic based design methodology, we conceive a space-efficient FFT architecture based on the use of a single reconfigurable vector of datapath units (rDPUs). A version of the conceived FFT is described in VHDL using the Quartus II system. Derived from algorithms for a software-based implementations on parallel computer systems [ACQUAH et al. 1987, AMOR et al. 1994, ARGUELLO et al. 1996, COOLEY AND TUKEY 1965, SWARTZTRAUBER 1987], the classical hardware design of the FFT is a matrix of DPUs where the input data is provided to the first column and then traverse the matrix column by column, following a butterfly connection scheme between column elements. In our approach, a single rDPU vector is implemented. In the first step, it executes the function of the first column over the input data. The data produced by the first iteration is then fed back to the inputs while the vector is reconfigured to emulate the second column. The process is iterated until the FFT is done.

Closeness of rewriting specifications to algebraic equations simplifies the verification of the logical soundness of the modeled solutions. This approach provides an informal preview of solutions that are expected to be space and power efficient, scarifying speed. But to be conclusive, the conceived systems should be implemented. Although the discussion follows mainly the lines of this interesting case study, we point out that the proposed rewriting-logic specification methodology is general and not restricted to the case of reconfigurable systolic architectures. This generality is made evident by the recent conception of other two relevant architectures:

- First, in [AYALA-RINCÓN et al. 2002] a speculative processor is specified by rewriting rules whose buffer and speculation control was guided by logic strategies. Soundness of this specification was proved by hand based on rewriting theory, but it can be mechanized by translating this rewriting specification to logical theories over proof assistant systems such as PVS [OWRE et al. 1995], which is our target in a current research.
- Second, in [AYALA-RINCÓN et al. 2004] it is conceived a dynamically rDPA, which is adapted by simple reconfigurations to the treatment of different combinatorial problems over words such as local and global sequence alignment, approximate string matching and longest common subsequences (all relevant in molecular processing). This rDPA implements a general *dynamic-programming*[1] algorithm, in which the components of the *dynamic-programming table* are computed by a recursive dependency on the previous row, column and diagonal components of the table. Reconfiguration of the rDPA adjusts its operations and interconnections for computing the recursive dependency which solves any of these specific problems. In this setting also dynamic reconfiguration is of interest since, for example, space-adequate algorithmic solutions for local sequence alignment are based in detecting high scores of local alignments, without explicitly computing the whole dynamic-programming table, and then applying the global alignment solution over the regions of interest. For the systolic array this corresponds to a dynamic reconfiguration changing the parameters from local to global alignment as well as the direction of processing the sequences.

The organization of the paper is described as follows. Section 2 provides historical remarks on the application of rewriting and rewriting-logic in hardware modeling and design and basic concepts on rewriting and rDPAs. Additionally, it presents the appropriateness of rewriting-logic in modeling reconfigurable systems and rDPAs from a general point of view. Section 3 exemplifies the rewriting-based specification and simulation methodology for simple rDPAs used for implementing 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 non trivial case study of the FFT. Section 5 shortly describes the specification of our space efficient implementation of this FFT in VHDL using the Quartus II system and Section 6 is the conclusion.

---

[1] The *dynamic-programming method* is an important and well-known technique for the design of efficient algorithms, that essentially detects and avoids redundancies of recursive solutions by computing a *dynamic-programming table* in which results of recursive calls are stored avoiding in this way redundant computations [CORMEN el al 2001, BAASE AND VAN GELDER 1999].

2. BACKGROUND

We include historical remarks on the use of rewriting in the architectural context, the minimal needed notions on rewriting and rDPAs and discuss the appropriateness of rewriting-logic in the specification of systolic and reconfigurable architectures. For a detailed presentation on rewriting see [BAADER AND NIPKOW 1998] and, specifically, on rewriting-logic see the selected works in [MARTÍ-OLIET AND MESEGUER 2002a] (In particular, the roadmap on rewriting-logic in [MARTÍ-OLIET AND MESEGUER 2002b]) and for systolic arrays see [KUNG AND LEISERSON 1978, KUNG 1987].

2.1 Historical Remarks on the Application of Rewriting(-Logic) in Hardware

After the seminal work of Knuth-Bendix about the *completion* of algebraic equational specifications [KNUTH AND BENDIX 1970], *Term Rewriting Systems (TRS)* have 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. Rewriting theory conform an adequate theoretical framework for reasoning about the functional programming paradigm and implementation of functional languages. Nowadays, rewriting is being promoted as a new programming paradigm of great usefulness because of its high level of abstraction which is a consequence of the simplicity of its operational semantics: *matching* and *substitution* [KIRCHNER AND MOREAU 2001].

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 KAPUR AND SUBRAMANIAM [1997,2000], [KAPUR 2000] as well as Arvind's group that treated the specification of processors over simple architectures [ARVIND AND SHEN 1999], SHEN AND ARVIND [1998A,1998B], the rewrite-based description and synthesis of simple logical digital circuits [HOE AND ARVIND 1999] and the description of cache protocols over memory systems [SHEN et al. 1999]. In this context TRS use terms and rules to describe hardware states and behavior, and its detailed semantics make it a good choice for debugging complex and highly concurrent designs. Unlike other high-level hardware description language (HDL) approaches, TRS do not use software expressivity to describe circuit behavior.

Rewriting-logic, that extends the pure rewriting paradigm allowing for logical control of the application of the rules by logic strategies, has been showed of greater flexibility than purely rewriting for discriminating between fixed and reconfigurable elements of reconfigurable architectures. This allows a natural and quick conception and simulation of implementations in configware, directly, or, for rapid hardware prototyping. 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 [AYALA-RINCÓN et al. 2002]. To achieve this we applied rewriting-logic allowing a logical control of the application of the rules by strategies [MESEGUER 2000, BOROVANSKÝ et al. 2002].

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 MARTÍ-OLIET AND MESEGUER [2002a, 2002b].

2.2 Rewriting Theory

A Term Rewriting System, TRS for short, is defined as a triplet $\langle R, S, S_0 \rangle$, where S and R are respectively sets of *terms* and of *rewrite rules* of the form $l \rightarrow r$ ***if*** p 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 [ARVIND AND SHEN 1999], 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 "$\rightarrow$" and its reflexive-transitive closure by "$\rightarrow^*$".

In the architectural context two states (or configurations) s and t are related by the reflexive-transitive closure: $s \rightarrow^* t$ whenever it is possible to reach the configuration t from the configuration s according to the rewriting rules (transitions) of the system.

The important notions of *termination* and *confluence* correspond to practical computational aspects as the finiteness of processes and their determinism:

- a TRS is *terminating* if there are no infinite sequences of the form $s_0 \rightarrow s_1 \rightarrow ...$
- a TRS is *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 $t_2 \rightarrow^* u$.

Satisfaction of these two properties is important in the algorithmic context because they guarantee for effective computation of unique answers: assuming termination and confluence there is always possible to *normalize* any term s reaching its unique normal form in a finite number of rewriting steps: there exists a sole normal form $\underline{s}$ such that $s \rightarrow^* \underline{s}$. Although in the architectural context termination is not a necessary property; for instance, a processor should be able to run infinitely according to the charged procedure. Consequently, a well known collection of results in rewriting theory related with confluence of non-terminating TRSs apply in this context.

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 $s \rightarrow^* t$.

2.3 Appropriateness of Rewriting-Logic for Modeling Reconfigurable Systems

Using these rewriting notions 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 logic control of the ordering in which rules are applied. Rewriting jointly with logic is known as *rewriting-logic*. For a simple example of the use of this logic control, consider the following labeled rules for the greater common divisor *gcd*:

$$[\textbf{subs}]\ \ gcd(m,n) \rightarrow gcd(m-n,n)\ \ \textbf{\textit{if}}\ (m > n \text{ and } n > 0)$$

$$[\mathbf{swit}]\ \ gcd(m,n)\ \rightarrow gcd(n,m)\ \textbf{\textit{if}}\ \ (m \neq n \text{ and } n > 0)$$

Firstly, observe that this TRS is non terminating because of the second rule: $gcd(x,y) \rightarrow_{swit} gcd(y,x) \rightarrow_{swit} gcd(x,y) \rightarrow_{swit} ...$, for any $x\neq 0$, $y\neq 0$. Secondly, consider a *logic strategy* for applying the rules of the form *"apply the rule* **subs** *as many times as possible and then apply once the rule* **swit** *and repeat this as much as possible"*, which can be written as the regular expression (**subs*** **swit**)*. Notice that "subs*" means "normalize with the rule **subs**". By applying this logic strategy we obtain a rewriting mechanism for effectively computing *gcd*. For instance, by applying these rules under this strategy we obtain the reduction: $gcd(96,126) \rightarrow_{swit} gcd(126,96) \rightarrow_{subs} gcd(30,96) \rightarrow_{swit} gcd(96,30) \rightarrow_{subs} gcd(66,30) \rightarrow_{subs} gcd(36,30) \rightarrow_{subs} gcd(6,30) \rightarrow_{swit} gcd(30,6) \rightarrow_{subs} gcd(24,6) \rightarrow_{subs} gcd(18,6) \rightarrow_{subs} gcd(12,6) \rightarrow_{subs} gcd(6,6)$.

Formally, a rewriting-logic system may be conceived as a rewriting system jointly with a set of labels and strategies, which are regular expressions over the alphabet of label symbols. In a rewriting-logic system, rules are labeled and strategies let the user specify how the rules should be applied. Strategies are mechanisms for selecting transitions (applicable rules), from a nondeterministic spectrum, in specific states (configurations).

Rewriting-logic is of general and practical applicability in the context of specification of hardware since rewriting rules and logic may be naturally adapted for discriminately representing in the necessary detail many hardware elements involved in processors and processes. Logic strategies are useful for separating architectural elements in general being useful, in particular, for discriminating reconfiguration and execution instructions in reconfigurable systems [AYALA-RINCÓN et al. 2003]. Although our experiments are implemented in the particular rewriting-logic environment ELAN, it is relevant to remark here the generality of the proposed discipline of specification. Rewriting rules are used to model transitions between configurations of a system. Then rewriting rules can model instructions as well as reconfiguration transitions. The separation between these transitions is done by specific strategies which indicate when instruction transitions are to be executed and when reconfiguration is to be done.

More specifically and formally, a rewriting system can model the operational semantics of a reconfigurable system by two different sets of labeled rewriting rules for the instruction and the reconfiguration transitions:

$$[\mathbf{inst_1}]\ \ l_1 \rightarrow r_1\ \ \textbf{\textit{if}}\ \ p_1 ; \dots ; [\mathbf{inst_n}]\ \ l_n \rightarrow r_n\ \ \textbf{\textit{if}}\ \ p_n$$

$$[\mathbf{rcg_1}]\ \ g_1 \rightarrow d_1\ \ \textbf{\textit{if}}\ q_l ; \dots ; [\mathbf{rcg_m}]\ \ g_m \rightarrow d_m\ \ \textbf{\textit{if}}\ q_m$$

These rules guide changes in the elements of the architecture which are selected by *matching* their left-hand sides under specific configurations given by the corresponding conditions of the rules. These changes are made effective by *substituting* (instances) of left-hand sides by right-hand sides of the rules. For instance, when specifying a processor by rewriting rules, a rewriting rule can model the changes in the register file during execution. And other rewriting rule can be used for simulating the changes of the program counter of the processor during execution of branching instructions. A reconfiguration rule may guide changes in interconnections of components, for instance. Labels of the sets of instruction and reconfiguration rules are not necessarily mutually different; in this way rules for specific task are automatically associated. One can have discriminated subsets of instruction rules for charging a buffer, issuing rules, in/out instructions, etc. Thus strategies, which can be written as regular expressions over the alphabet of labels, guide the application of these rules. For instance, if the set of labels for instruction and reconfiguration rules are unitary $(\text{inst}_1^* \text{ rcg}_1)^*$ is a strategy which indicates repeat as much as possible: apply any instruction rule until it is no longer possible and then a sole step of reconfiguration. In other words normalize the system with the instruction rules and then reconfigure and do this as much as possible. In the context of ELAN this can be written as *repeat***(normalize(inst*$_1$*); rcg*$_1$*)*.

An additional important aspect of rewriting-logic computational environments is their powerful use of types. Rewriting rules should have left- and right-hand sides of the same sort. This implies that the transitions modeled by rewriting change the state of architectural elements of the same sort or class. The typing theory admits usually subtypes as is the case of ELAN. Jointly with strategies this is very adequate for modeling the behavior of systolic systems: a whole cycle in a systolic array corresponds to normalization by the rules of a specific sort. In this way, simulation of a systolic array is straightforwardly represented by rewriting-logic.

2.4 Systolic Arrays and Reconfigurable Systems

The rDPA is a kind of generalization of the (non-reconfigurable) *systolic array* [KUNG 1987], a hardwired 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. Methodologies for mapping applications to systolic arrays are also useful to develop mappings onto rDPAs. However, in contrast to systolic arrays, rDPAs are not restricted to applications with regular data dependencies, and, also a wide variety of non-NN interconnect schemes can be additionally included [NAGELDINGER et al. 2000].

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.

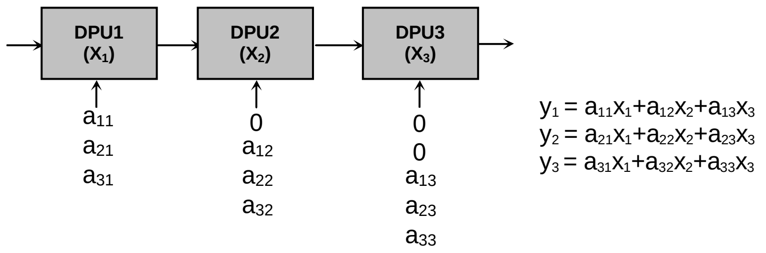

DPU1 (X_1)
DPU2 (X_2)
DPU3 (X_3)
a_{11} a_{21} a_{31}
0 a_{12} a_{22} a_{32}
0 0 a_{13} a_{23} a_{33}
$y_1 = a_{11}x_1+a_{12}x_2+a_{13}x_3$
$y_2 = a_{21}x_1+a_{22}x_2+a_{23}x_3$
$y_3 = a_{31}x_1+a_{32}x_2+a_{33}x_3$

Figure 1. Vector-matrix computation.

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. These limitations may be circumvented by using reconfigurable circuits, the most representative of them being the FPGA.

FPGAs provide fine grain reconfiguration working with bit wide operators. This kind of architecture provides high 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 specifies one among a set of available operations. The amount of configuration bits in this case is much less than in the fine grain FPGAs.

rDPA architectures (e. g. [HARTENSTEIN et al. 1995, 2001]) overcome the restriction of systolic arrays while keeping the benefits of a large degree of parallelism. In this approach, the operations performed by each rDPU 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* [STEIGER et al. 2004], which broadens even more the architectural alternatives. Here the *reconfiguration at run time* is driven by configware and should not be confused with classical software-driven switching at run time, like e. g. inside a CPU. 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, one could describe a reconfigurable system by its *configuration stream* and its data stream. While the instruction stream (software) is decoded and executed by an immutable hardware, the configuration stream (configware) changes the hardware organization itself. 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.

In the past we have not been aware, that in fact, the quarter century old systolic array is a paradigm shift. In contrast to the CPUs of the von Neumann paradigm the DPUs (Data Path Units) in a systolic array do not include a program counter nor a sequencing controller. Along with the systolic array the concept of data streams has been introduced ([KUNG, S. Y. 1987] and others), but usually hardly telling, how these multiple data streams entering and leaving systolic arrays are generated at run time. Such a distributed memory methodology has been popularized two decades later, mostly by Prof. Francky Catthoor and his group at IMEC ([CATTHOOR et al. 1998]): application-specific distributed memory architectures, also called asM (auto-sequencing memory [HARTENSTEIN et al. 1991]), where, instead of a program counter, data counters are used which reside in the memory banks [HERZ et al. 2002], which really means a new machine paradigm, the anti-machine paradigm - counterpart of the von Neumann paradigm (also see [HARTENSTEIN 2004]). The term *data streams* for multiple parallel data streams, which we have adopted from the area of systolic arrays, should not be confused with stream processing known from multimedia, dealing with *single streams of similar data objects* like known from video processing.

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. PROGRAMMING RECONFIGURABLE ARRAYS VIA REWRITING-LOGIC

For illustrating the use of rewriting-logic in this context, basic specifications of simple systolic arrays for vector and matrix multiplication are presented. In these systems each component -DPU as in Figure 1- is called a `MAC` (Multiplier/Adder). The modeling of the matrix/vector multiplier is presented in the Figure 1. The type definition for each `MAC` is shown in Table I and the structure of the systolic array in Figure 2.

Type definition in ELAN has the following syntax (Table I): 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 the operator where the '`@`' symbol 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 three `MAC's` and one `DataStream`. The `DataStream` is an object with three components of type `list[Data]`.

Table I. 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
```

The rule `sole` (Table II) 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, `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`. Afterwards, the multiplications between the contents of each first port `pi1` and the corresponding constant `ci` are placed in the first register of each `MAC`, 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, 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 changing the constants in each `MAC`. Then a computation with the 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`.

The Table III shows the rule `conf` created for reconfiguring the processor. It simply changes the contents of the constant part of the `MACs` (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).

Table II. ELAN description of the rule `Sole`

```
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
```

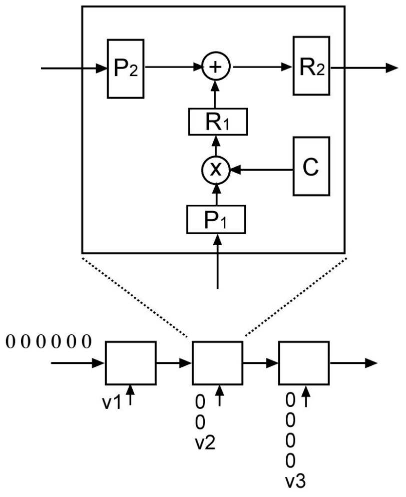


Figure 2. MAC systolic array architecture.

Table III. Rule `conf` of 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
```

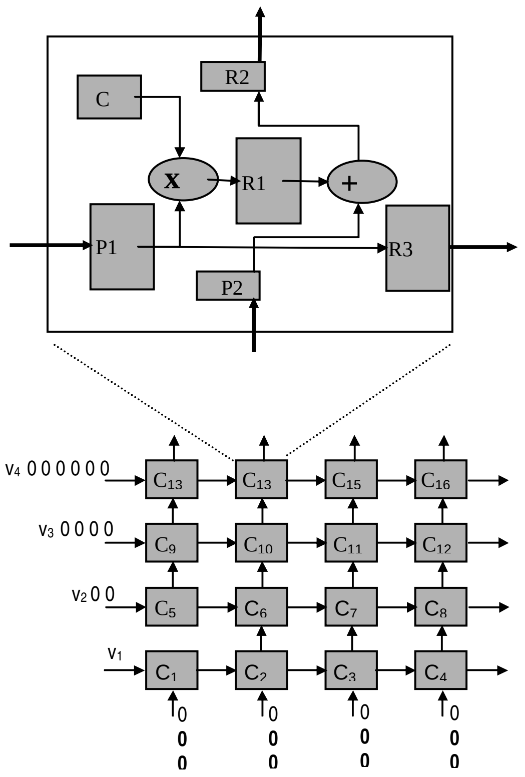


Figure 3. Systolic array for matrix multiplication.

The Figure 3 shows the structure of a systolic array for 4x4 matrix multiplication. Its description is given in the Table IV. 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`.

Table IV. A 4×4 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
```

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.

Table V. 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
   lW1,lW2,lW3,lW4,lS1,lS2,lS3,lS4    : 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]
   <  (lW1 lW2 lW3 lW4)
       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
      (lS1 lS2 lS3 lS4) >                =>
        <  (lW1 lW2 lW3 lW4)
           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
      (lS1 lS2 lS3 lS4) >
  end          ...
end
```

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

All these rules are very similar and one of them is presented in the Table V. 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 VI 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 VI. `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
```

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.

4. FFT MODELING WITH REWRITING-LOGIC

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

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

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.

The FFT is an *O(n ln n)* run time implementation of DFT based on a recursive algorithm proposed by [COOLEY AND TUKEY 1965]. This algorithm can be implemented in dataflow hardware as presented in classical text books on algorithms [CORMEN et al. 2001, BAASE AND VAN GELDER 1999, AKL 1997]. 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 4.

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 (×). 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.

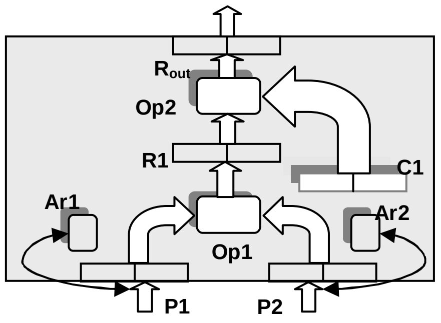
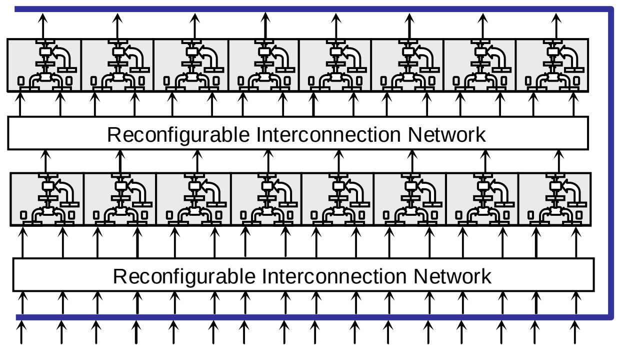


Figure 4. node architecture for FFT.

The Figure 5 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 4. 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.



Reconfigurable Interconnection Network
Reconfigurable Interconnection Network

Fig. 5. Two 8-array system.

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 $a_0+a_1\cdot x+\ldots+a_7\cdot x^7$ 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, ×; **1:** I,I,+,1, ×; **2:** I,I,+,1, ×; **3:** I,I,+,1, ×;
4: I,I,+,1, ×; **5:** I,I,+,1, ×; **6:** I,I,+,1, ×; **7:** I,I,+,1, ×;

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, it applies 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, $\times$;
4: 4,5,+,1, $\times$; **5:** 4,5,-,1, $\times$; **6:** 6,7,+,1, $\times$; **7:** 6,7,-, i, $\times$;

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, $a_2+\cdot a_6$, $(a_2-\cdot a_{6)i}$, a_1+a_3, a_1-a_3, $a_5+\cdot a_7$ and $(a_5-\cdot a_{7)i}$.
The third reconfiguration is given by the sequence:

0 **0:** 0, 2, +,1, $\times$; **1:** 1,3,+, 1, $\times$;
2: 0, 2, -, 1, $\times$; **3:** 1,3, -, 1, $\times$;
4: 4, 6, +,1, $\times$; **5:** 5,7,+, $(1+i)/\sqrt{2}$, $\times$;
6: 4, 6, -, i, $\times$; **7:** 5,7, -, $(-1+i)/\sqrt{2}$, $\times$;

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

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

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

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 VII. Notation "`<@ @> : ( num num ) complexUnit;`" means that "`< >`" is a binary operator of type `complexUnit` with two parameters of type `num`.

The 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 4.

Table VII. 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( @ )               : ( MAConfig ) recMAC;
 getMACInit( @,@,@ )          : ( int complexUnit complexUnit ) MAC;
 getMAC( @,@ )                : ( MAC MACsArray ) MAC;
 extractRegValue ( @ )        : ( MAC ) complexUnit;
 ( @ @ @ @ @ @ )              : ( int int num num Op Op ) MAConfig;
 <  @ @ @ @ @ @ @ @  >        : ( MAConfig MAConfig MAConfig MAConfig
                                  MAConfig MAConfig MAConfig MAConfig)
                                                     ReconfParameter;
 continue                     : ReconfParameter; //vacuous reconfiguration
 <  @ @ @ @  >                : ( int list[ReconfParameter] MACsArray
                                  MACsArray ) Proc;
end
```

Each simultaneous execution-reconfiguration step of this system is specified by rewriting rules as the one presented in the Table VIII. 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 VIII. 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
```

Table IX. 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 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.

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 IX. 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.

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 X. 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 `MAConfig` (see the Table VII). 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.

Table X. Rule of dynamic reconfiguration

```
[] reconfigure(MACsArray(
      [ fix0 # rec0 ] [ fix1 # rec1 ] [ fix2 # rec2 ] [ fix3 # rec3 ]
      [ fix4 # rec4 ] [ fix5 # rec5 ] [ fix6 # rec6 ] [ fix7 # rec7 ] ),
    < MAConfig0 MAConfig1 MAConfig2 MAConfig3
      MAConfig4 MAConfig5 MAConfig6 MAConfig7 >)
   =>
  MACsArray([fix0 # getRecMAC(MAConfig0)][fix1 # getRecMAC(MAConfig1)]
            [fix2 # getRecMAC(MAConfig2)][fix3 # getRecMAC(MAConfig3)]
            [fix4 # getRecMAC(MAConfig4)][fix5 # getRecMAC(MAConfig5)]
            [fix6 # getRecMAC(MAConfig6)][fix7 # getRecMAC(MAConfig7)])

 end
```

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 VIII). 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
```

`PROCESS` basically organizes the application of rules for propagating the input data and reconfiguration stream, repeating the `oneCycle` rules (see the Table VIII) 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 [BAASE AND VAN GELDER 1999].

4.3 A physical in-place implementation of the 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 from the theoretical perspective of complexity of algorithms.

From the practical point of view, this is possible since computing operations over (large) 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 single 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 single 8-array system, which is the one treated in the next section, is possible to implement the FFT alternating reconfigurations and execution 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 (however it is possible to use a unique `MAC`, notice that our considerations are for parallel processing in run time *O(ln(n))*) such as the well-known *in place* algorithmic solutions of the FFT [AYALA-RINCÓN et al. 2003]. Of course, in this single 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 single 8-array architecture was modeled and simulated in ELAN, using a similar approach. Both ELAN implementations for the single and two 8-array architectures are available in www.mat.unb.br/~ayala/TCgroup. Two alternatives for modeling the reconfiguration are provided: built-in reconfiguration (which fixes the applicability of the system for computing FFT) and reconfiguration given as a stream of data by the user. The latter one allows for application of the systems for other computations different from FFT. For this alternative, the input includes data and reconfiguration parameters. For computing the FFT the input looks as:

```
inputFFT  (  < a0 >, < 0 >,   < a4 >, < 0 >,
             < a2 >, < 0 >,   < a6 >, < 0 >,
             < a1 >, < 0 >,   < a5 >, < 0 >,
             < a3 >, < 0 >,   < a6 >, < 0 >  )
hardware_configuration (
<(0 0 1,0000 0,0000 + *)(0 0 1,0000 0,0000 + *)
 (0 0 1,0000 0,0000 + *)(0 0 1,0000 0,0000 + *)
 (0 0 1,0000 0,0000 + *)(0 0 1,0000 0,0000 + *)
 (0 0 1,0000 0,0000 + *)(0 0 1,0000 0,0000 + *)>.
<(0 1 1,0000 0,0000 + *)(0 1 1,0000 0,0000 - *)
 (2 3 1,0000 0,0000 + *)(2 3 0,0000 1,0000 - *)
 (4 5 1,0000 0,0000 + *)(4 5 1,0000 0,0000 - *)
 (6 7 1,0000 0,0000 + *)(6 7 0,0000 1,0000 - *)>.
<(0 2 1,0000 0,0000 + *)(1 3 1,0000 0,0000 + *)
 (0 2 1,0000 0,0000 - *)(1 3 1,0000 0,0000 - *)
 (4 6 1,0000 0,0000 + *)(5 7 0,7071 0,7071 + *)
 (4 6 0,0000 1,0000 - *)(5 7 minus(0,7071) 0,7071 - *)>.
<(0 4 1,0000 0,0000 + *)(1 5 1,0000 0,0000 + *)
 (2 6 1,0000 0,0000 + *)(3 7 1,0000 0,0000 + *)
 (0 4 1,0000 0,0000 - *)(1 5 1,0000 0,0000 - *)
 (2 6 1,0000 0,0000 - *)(3 7 1,0000 0,0000 - *)>.continue.nil)
```

The first part of the input corresponds to the polynomial coefficients to be processed and the second part to the reconfiguration parameters for computing the FFT. Notice that complex numbers are represented as two four decimal numbers for their real and complex parts, which means this ELAN modeling goes to this level of description.

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. VHDL DESIGN OF THE RECONFIGURABLE FFT ARCHITECTURE

Rewriting-logic allows the designer to explore the architectural design space in a very high abstraction level. The models obtained provide information about how the system works and can be used for verification purposes as well as rough performance estimation. In the FFT example, the adoption of one or two FFT rows may be analyzed based on the simulation performed through the rewriting rules and logic strategies. Processing and reconfiguration steps are alternated and the TRS model shows how many phases are needed, and how data should be feedback and transmitted in order to compute the correct results. Clearly, the use of a single row of MACs demands more processing steps to accomplish the task. It is obvious that two rows take more space than one, but the cost of the steering logic required in each case is not that obvious to determine. The target technology should be taken into account. For FPGAs, the restricted amount of interconnection resources may produce unexpected results, since logic elements may become unreachable if too much interconnections are used and the size in terms of FPGA surface may grow more than expected. For full custom design, the steering logic can be better adapted to the circuit architecture.

Considering hardware resources needed to implement the design, the operators such as adders and multipliers are explicitly presented in the rules. Although in pure rewriting-logic no ordering is implied by the rules, the strategies allow the “sequencialization” of the tasks and, thus, provide information about possible resource sharing. The steering logic, on the other hand, can only be guessed by the fanin and fanout of the ports, since the interconnections are defined by the position of the variables inside the rewriting expressions.

The TRS description is a formal starting point to the physical implementation of the design. Although very abstract, this first step in fundamental in the design process, since the detection of errors in the system specification and in early design phases is fundamental to reduce the design cycle time and test and debugging costs.

Here we show how the single row FFT modeling can be used as starting point to FPGA prototyping. The rewriting system described in subsection 4.3 validated the operations and reconfiguration steps needed to implement a single reconfigurable row of MACs. Thus, the function to be executed by each MAC could be obtained by analyzing the rules of the TRS description. The I/O of the modules, in this case, was explicitly represented and can be easily figured out. The interconnections are extracted from the rules by checking the fanin and fanout of the ports specified in the rules. Since this prototype targets FPGAs, the natural candidate to implement steering logic is the multiplexor.

The general structure of the single row reconfigurable FFT derived from the TRS description is presented in the Figure 6. The ROM stores the reconfiguration data, extracted from the TRS rules. The simple FSM that control the iterations is obtained by analyzing the strategies that control the application of the rules. In this way, all necessary information to design the single row FFT were inferred from the rewriting rules and logic strategies.

The subsections that follow give some details of the hardware implementation.

5.1 The functional units implementation

The FFT manipulates complex variables, stored in register pairs for the real and imaginary elements.

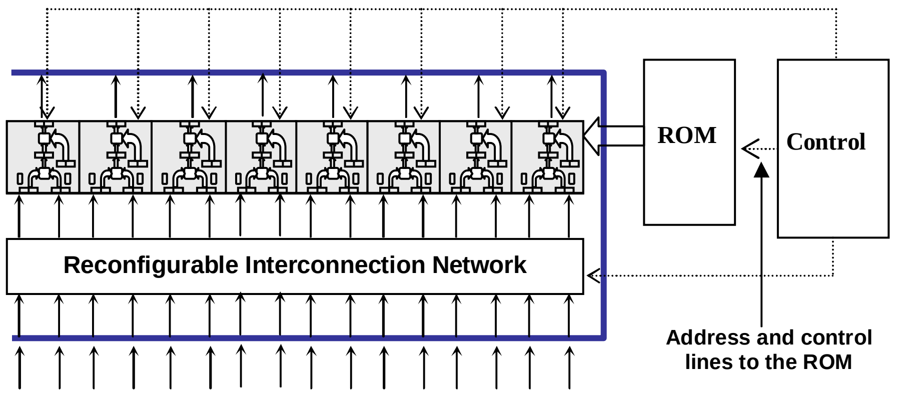

ROM
Control
Reconfigurable Interconnection Network
Address and control lines to the ROM

Figure 6. The architecture of the FFT circuit

In the same way the functional units need to deal with complex operands. The operation in the functional units is selected by two bits as showed in the Figure 7a. Figure 7b shows the four multipliers required to compute the complex multiplication.

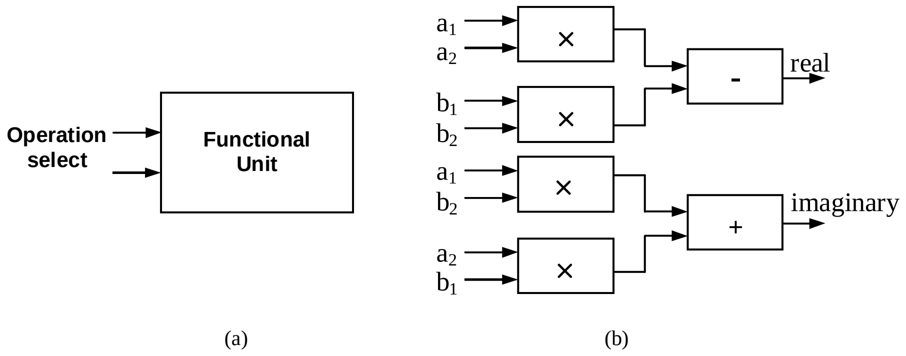


Figure 7. (a) The functional unit block. (b) The complex multiplier

5.2 The reconfigurable interconnection network (the butterfly operation)

The butterfly is a basic operation in the FFT computation which produces a pair of complex values in a stage *m* from a pair or values from the preceding stage *m–1* following the equations:

$X_m[n] = X_{m-1}[n] + X_{m-1}[n + r]$ and $X_m[n + r] = X_{m-1}[n] – X_{m-1}[n + r]$

where $r = N/2^m$ and $m = 1,..., \log_2 N$.

For the 8-array FFT the sequence of the butterfly operation for each step, obtained from the TRS description, is described in the Table XI.

Table XI. Mapping the interconnections on the FFT steeps

	Input Port	St_0	St_1	St_2	St_3
MAC0	P1	Extern	MAC0	MAC0	MAC0
	P2	Extern	MAC1	MAC2	MAC4
MAC1	P1	Extern	MAC0	MAC1	MAC1
	P2	Extern	MAC1	MAC3	MAC5
MAC2	P1	Extern	MAC2	MAC0	MAC2
	P2	Extern	MAC3	MAC2	MAC6
MAC3	P1	Extern	MAC2	MAC1	MAC3
	P2	Extern	MAC3	MAC3	MAC7
MAC4	P1	Extern	MAC4	MAC4	MAC0
	P2	Extern	MAC5	MAC6	MAC4
MAC5	P1	Extern	MAC4	MAC5	MAC1
	P2	Extern	MAC5	MAC7	MAC5
MAC6	P1	Extern	MAC6	MAC4	MAC2
	P2	Extern	MAC7	MAC6	MAC6
MAC7	P1	Extern	MAC6	MAC5	MAC3
	P2	Extern	MAC7	MAC7	MAC7

For example, in the step 0 the operands for the two ports (for each MAC) are the coefficients of the input polynomial (see section 4) just as described in column 1 of the Table XI (external inputs). In the following step the butterfly operation defines the new sources of the operands for each MAC. For example, in the step 0 for the MAC0 receives the inputs from MAC0 (in the port P1) and MAC1 (in the port P2). Thus the butterfly operation represents a reconfiguration step of the interconnections in the 8-array. The reconfigurable interconnection network was implemented using 4×1 multiplexers. The Figure 8 shows the multiplexer interconnection for routing the data to the input ports in a MAC.

5.3 The control unit

The operation of the modules integrating the whole of the system is scheduled by a 12-state synchronous FSM. In order to execute each step of the FFT algorithm the FSM execute the *propagate*, *reconfigure* and *execute* states. The *propagate-state* deals with the control lines of the multiplexer and controls de enable lines for the inputs/outputs registers.

The *reconfigure-state* controls the ROM memory address lines in order to configure the functional units and all of the constant registers. The *execute-state* controls the port out enable lines to obtain the result for each steep.

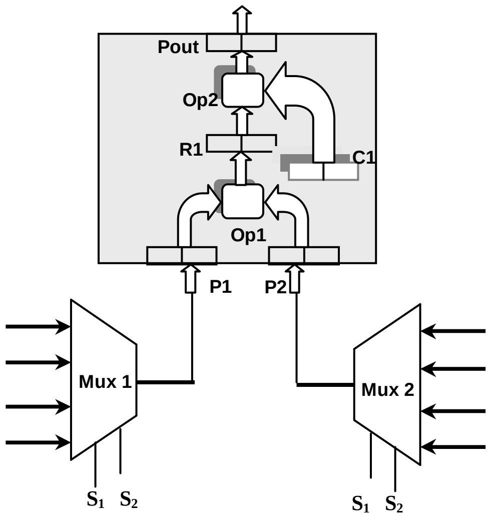


Figure 8. Implementation of the interconnection network

5.4 Area and running time results

The FFT circuit was implemented as a hierarchic description. The MAC array and the interconnection network implement the FFT-processor module. The top of the design consists of the FFT-processor, the ROM and the control unit. Each module of the architecture was implemented and tested for validation. The modules were coded using parameterized RTL-level VHDL allowing the reusability and easy configuration. The complete description was compiled and simulated in the Quartus II system from Altera Corporation [2004]. The area and timing results for a 8 bits length words coefficients using an APEX EP20K400 as target device are showed in the Table XII.

The computation of one step of the FFT takes 2 cycles, one cycle to load data into the input registers and compute de complex result and a second cycle to store the result in the MAC output registers, to feedback the results through the reconfigurable network and to reconfigure the MAC operations. The design presented here receives as inputs 8 coefficients. The complete FFT is performed in 3 steps. Thus, we have that the global time to produce the outputs is 29,64 / 6 = 4,94 millions of FFT by second.

Table XII. Synthesis Results

Feature	Reconfigurable FFT	Pipelined FFT
Family	APEX	APEX
Device	EP20K400FC672-2XV	EP20K400FC672-2XV
Operating frequency	29,64 MHz	35,11 Mhz
Logic elements number	1340 (8%)	10820 (65%)
Delay	33.7 ns	28,48 ns
Throughput	4,94 M FFT/s	35,11 M FFT/s

For comparison purposes we implemented a pipeline version of the FFT, which is built using the same structure of the reconfigurable FFT but without feedback. It is able to compute one FFT by cycle after the pipe is filled. The synthesis results are shown in the last column of the Table XII. As expected, the pipelined version produces faster results, around 7 times the throughput of the reconfigurable FFT but consuming around 8 times more hardware resources.

6. CONCLUSIONS

Rewriting-logic is a powerful tool for high abstraction level modeling and simulation of integrated systems. In this work it was shown how rewriting rules and rewriting-logic strategies can be used to model reconfigurable architectures and in particular reconfigurable systolic arrays, which are architectures that cover a broad range of applications providing massive parallelism. Representing the reconfiguration capabilities of hardware in this way, that is by logic strategies, outside of the operational semantics of the rewriting rules, seems 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 (see for example [BJESSE et al. 1998]). But we believe that the proposed specification approach is more adequate than those purely rewriting or functional based approaches. In fact, in our proposed rewriting-logic based setting, we showed how we can naturally profit from the discrimination provided by logical strategies to separate execution from reconfiguration operations simplifying the purely rewrite based specification, experimentation, simulation (and even verification [AYALA-RINCÓN et al. 2002]) of reconfigurable systems. Even sophisticated mechanisms of reconfiguration such as dynamic reconfiguration and self-reconfiguration appear to be straightforwardly conceivable via rewriting rules and 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 operations. For the implementation of complex operators such as the FFT, we have conceived physical systems, which are running time efficient ($O(\ln n)$) 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 [NAGELDINGER 2001].

The VHDL description of our space efficient FFT ELAN base model was developed taking into account its possible reuse for different applications, like matrix multiplication, convolution, and signal processing algorithms in general. All of the modules of the circuit were parameterized for easy reuse and configuration. One of the advantages of this approach is that this reconfigurable module can fit into systems on chip with area and reconfigurable resource restrictions and still produce high throughput. A course grain implementation of this architecture could be suitable for a variety of signal processing problems with a much higher throughput than that obtained with standard FPGAs or by software.

Future work will address new reconfiguration schemes, since the reconfiguration time in this case is constrained by the ROM access time. Alternatives like a pipeline between reconfiguration will be studied. Currently, to study the possibilities of dynamic reconfiguration more sophisticated models are under development. In particular, modeling and design of adequate dynamically reconfigurable architectures for the efficient treatment of dynamic programming based solutions of several problems such as local and global sequence alignment, longest common subsequence and approximate string matching are presented in [AYALA-RINCÓN et al. 2004]. Also, we develop a tool for intelligently translating ELAN specifications into PVS theories to allow the semi-automated verification using formal methods instead of manual verification (and requiring simulation). This integration includes strategies for applying rewriting proving methods such as the Knuth-Bendix-Huet critical pair lemma, detection of critical pairs, verification of joinability, etc. in the system PVS [OWRE et al. 1995]. Additionally, we develop a tool interfacing correct ELAN specifications of algebraic operators into the KressArray Xplorer for hardware design space exploration. The current version of this tool is targeted for the Pact eXtreme Processing Platform (XPP), with support for additional architectures planned in future versions [MORRA et al. 2005].

ACKNOWLEDGMENTS

We would like to thank Rodrigo Borges Nogueira for implementing the ELAN prototypes.

REFERENCES

ABNOUS, A. AND SENO, K., AND ICHIKAWA, Y. AND WAN, M., AND RABAEY, J.M. 1998. Evaluation of a Low-Power Reconfigurable DSP Architecture. In *Proceedings of the 5th Reconfigurable Architectures Workshop*, IPPS/SPDP '98 Workshops, Orlando, Florida, USA, 1998, pp. 55-60.

ACQUAH, J. AND RICE, M. D. 1987: Constant Geometry Algorithms for Computing Fourier Transforms on Transputer Arrays; Melpar - E-Systems Division, Fairfax, Virginia, December, 1987

AKL, S.G. 1997. *Parallel Computation: Models and Methods.* Prentice-Hall, 1997.

ALTERA® Corporation. *Quartus II User Guide.* Available at http://www.altera.com. Accessed in 2004.

AMOR, M., AND M. J. MARTIN, D. BLANCO, O. PLATA, F. F. RIVERA AND F. ARGUELLO 1994: Vectorization of the Radix r Self-Sorting FFT; Parallel Processing: CONPAR94/ VAPP VI, Linz, Austria, September 6-8, 1994, pp. 208-217, Springer-Verlag, Berlin, Germany, B. Buchberger and J. Volkert, Eds., LNCS no. 854, 1994.

ARGUELLO, F. AND M. AMOR AND E.L. ZAPATA 1996: FFTs on Mesh Connected Computers; Parallel Computing, vol. 22, no. 1, January 1996, pp. 19-39

ARVIND AND SHEN, X. 1999. Using Term Rewriting Systems to Design and Verify Processors. *IEEE Micro 19(3)*: pp. 36-46.

AYALA-RINCÓN, M., AND NETO, R.M., AND JACOBI, R.P., AND LLANOS, C.H., AND HARTENSTEIN, R.W. 2002. Applying ELAN Strategies in Simulating Processors over Simple Architectures. *In roceedings 2nd Reduction Strategies in Rewriting and Programming, Elsevier Electronic Notes in Theoretical Computer Science 70(6)*: pp. 1-16.

AYALA-RINCÓN, M., AND NOGUEIRA, R.B., AND JACOBI, R.P., AND LLANOS, C.H., AND HARTENSTEIN, R.W. 2003. Modeling a Reconfigurable System for Computing the FFT in Place via Rewriting-Logic. *In Proc. 16th Symposium on Integrated Circuits and System Design (SBCCI'03),* IEEE CS, pp. 205-210.

AYALA-RINCÓN, M. AND JACOBI, R. P. AND CARVALHO, L.G.A. AND LLANOS, C.H. AND HARTENSTEIN, R, 2004. Modeling and Prototyping Dynamically Reconfigurable Systems for Efficient Computation of Dynamic Programming Methods by Rewriting-Logic. *In Proc. 17th Symposium on Integrated Circuits and System Design (SBCCI'04),* ACM, 248-253. Journal version to appear in *Genetic and Molecular Research.*

BAADER, F. AND NIPKOW, T, 1998. *Term Rewriting and all That.* Cambridge University Press.

BAASE, S., AND VAN GELDER, A. 1999. *Computer Algorithms: Introduction to Design and Analysis.* Addison-Wesley.

BECKER, J., AND HARTENSTEIN, R.W. 2003. Configware and morphware going mainstream. *Journal of Systems Architecture 49*:127-142.

BECKER, J. AND HARTENSTEIN, R.W. AND HERZ, M. AND NAGELDINGER, U. 1998. Parallelization in Co-Compilation for Configurable Accelerators. *In Proc. Asia and South Pacific Design Automation Conference (ASP-DAC)*, Yokohama, Japan, pp. 23-33.

BJESSE, P. AND CLAESSEN, K. AND SHEERAN, M. AND SINGH, S, 1998. Lava: Hardware Design in Haskell. In *Proc. of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, SIGPLAN Notices 34(1), 1999, ACM, 174-184.

BOROVANSKÝ, P. AND KIRCHNER, C. AND KIRCHNER, H., AND MOREAU, P.-E, 2002. ELAN from a rewriting logic point of view. *Theoretical Computer Science 285(2),* pp. 155-185.

CATTHOOR, F. AND WUYTACK, S. W., AND DEGREEF, E., AND BALASA, F., AND NACHTERGAELE, L., AND VANDECAPPELLE, A, 1998. Custom Management Methodology; Kluwer Academic Publishers (now: Springer), Boston, MA, 1998.

CIRSTEA, H. AND KIRCHNER, C, 2000. Combining Higher-Order and First-Order Computation Using rho-Calculus: Towards a Semantics of ELAN. In *Frontiers of Combining Systems 2*, Studies on Logic and Computation 7. D.M. GABBAY AND M. DE RIJKE, Eds., Research Studies Press/Wiley, 2000, pp. 95-121.

CLAVEL, M. AND DURAN, F. AND EKER, S. AND LINCOLN, P. AND MARTI-OLIET, N. AND MESEGUER, J. AND QUESADA, J. F, 2002. Maude: specification and programming in rewriting logic. *Theoretical Computer Science 285(2),* pp. 187-243.

COOLEY, J. W. AND J. W. TUKEY 1965: An algorithm for the machine calculation of complex Fourier series; Math. Comput., vol. 19, no. 4, pp. 297-391, 1965

CORMEN, T.H., AND LEISERSON, C.E., AND RIVEST R. L., AND STEIN, C. 2001. *Introduction to Algorithms*, The MIT Press, 2001.

DIACONESCU, R., AND FUTATSUGI, K. 2002. Logical foundations of CafeOBJ. *Theoretical Computer Science 285(2),* pp. 289-318.

HARTENSTEIN, R. AND HIRSCHBIEL, A. G. AND RIEDMUELLER, M. AND SCHMIDT, K. AND WEBER M, 1991. (2nd best paper award): A High Performance Machine Paradigm Based on Auto-Sequencing Data Memory; HICSS-24 (24th Hawaii Int'l Conference on System Sciences), Koloa Hawaii.

HARTENSTEIN, R., AND KRESS, R., AND REINIG, H. 1995. A Scalable, Parallel and Reconfigurable Datapath Architecture. *In Proceedings 6th Int. Symposium on IC Technology, Systems and Applications (ISIC'95)*, Singapore.

HARTENSTEIN, R. 2001 (invited embedded tutorial): Coarse Grain Reconfigurable Architecture; Proc. Asia and South Pacific Design Automation Conference 2001 (ASP-DAC 2001), Yokohama, Japan, January 30 - February 2, 2001.

HARTENSTEIN, R. 2004 (invited paper): The Digital Divide of Computing; 2004 ACM International Conference on Computing Frontiers (CF04); April, 14-18, 2004. Ischia, Italy.

HERZ, M., AND MIRANDA, S., AND BROCKMEYER, M., AND CATTHOOR, F. AND R. HARTENSTEIN 2002: (invited paper): Memory Organization for Stream-based Reconfigurable Computing; 9th IEEE International Conference on Electronics, Circuits and Systems (ICECS 2002), September 15-18, 2002, Dubrovnik, Croatia.

HOE, J. C. AND ARVIND 1999. Hardware Synthesis from Term Rewriting Systems. In *Proc. of the 10th IFIP Int. Conf. on VLSI* - VLSI'99, Kluwer (now: Springer), 595-619.

KAPUR, D. 2000. Theorem Proving Support for Hardware Verification. In *3rd Int. Workshop on First-Order Theorem Proving.* Invited talk, St. Andrews, Scotland, 2000.

KAPUR, D. AND SUBRAMANIAM, M, 1997. Mechanizing Verification of Arithmetic Circuits: SRT Division. In *Proceedings 7th Foundations of Software Technology & Theoretical Computer Science* (FST&TCS). Springer-Verlag LNCS, Vol. 1346, 103-122.

KAPUR, D. AND SUBRAMANIAM, M. 2000. Using and Induction Prover for Verifying Arithmetic Circuits. *Journal of Software Tools for Technology Transfer 3(1)*:32-65.

KIRCHNER, H. AND MOREAU, P.-E. 2001. Promoting Rewriting to a Programming Language: A Compiler for Non-Deterministic Rewrite Programs in Associative-Commutative Theories. *Journal of Functional Programming* 11(2): pp. 207-251.

KNUTH, D. E. AND BENDIX, P. B, 1970. Computational Problems in Abstract Algebra. In *Simple Word Problems in Universal Algebras*. J. LEECH, Ed., Pergamon Press, Oxford, 263-297.

KUNG, H.T. AND LEISERSON, C. E. 1979. Systolic Arrays for VLSI. In *Sparse Matrix Proc. 1978*. Society for Industrial and Applied Mathematics, pp. 256-282, 1978.

KUNG, S. Y. 1987. *VLSI Array Processors*. Prentice-Hall, 1987.

MARTÍ-OLIET, N. AND MESEGUER, J. eds., 2002A. Special issue on Rewriting Logic and its Applications. *Theoretical Computer Science 285(2),* pp. 119-564.

MARTÍ-OLIET, N. AND MESEGUER, 2002B. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science 285(2),* pp. 121-154.

MESEGUER, J. 2000. Rewriting Logic and Maude: Concepts and Applications. In Eleventh Int. Conf. on Rewriting Techniques and Applications RTA 2000, L. Bachmair Ed., LNCS, Vol. 1833, pages 1-26, Springer-Verlag, 2000.

MESEGUER, J. 2003. Executable Computational Logics: Combining Formal Methods and Programming Language Based System Design. *In ACM and IEEE Proc. First Int. Conf. on Formal Methods and Models for Co-design* (MEMOCODE'03), IEEE CS, 3-12.

MORRA, C. AND BECKER, J. AND AYALA-RINCON, M. AND HARTENSTEIN, R, 2005. FELIX: Using Rewriting-Logic for Generating Functionally Equivalent Implementations. In *Proc. 15th International Conference on Field-Programmable Logic and Applications* (FPL 2005). IEEE CS, pp. 25-30.

NAGELDINGER, U. 2001. *Coarse-Grained Reconfigurable Architecture DesignSpace Exploration*. Ph. D. Thesis, Technische Universität Kaiserslautern, 2001.

NAGELDINGER, U. AND HARTENSTEIN, R. AND HERZ, M., AND HOFFMANN, T. 2000. Kress Array Explorer: A New CAD Environment to Optimize Reconfigurable Datapath Array Architectures. In *Proceedings 5th Asia and South Pacific Design Automation Conference* - (ASP-DAC 2000), Yokohama, Japan, pp. 163-168, 2000.

OWRE, S. AND RUSHBY, J. M. AND SHANKAR, N. AND SRIVAS, M. K, 1995. A tutorial on using PVS for hardware verification. In *Theorem Provers in Circuit Design: Theory, Practice, and Experience*, LNCS, Vol. 901, pages 258-279, SpringerVerlag.

PACTCORP http://pactcorp.com, accessed in 2005.

SHEN, X. AND ARVIND, 1998A. Design and Verification of Speculative Processors. Technical Report 400A, *Laboratory for Computer Science - MIT*. Also in the Proc. of the Workshop on Formal Techniques for Hardware and Hardware-like Systems, June 1998, Marstrand, Sweden.

SHEN, X. AND ARVIND 1998B. Modeling and Verification of ISA Implementations. Technical Report 400B, *Laboratory for Computer Science - MIT*. Also in the Proc. of the Australasian Computer Architecture Conference, February 1998, Perth, Australia.

SHEN, X., AND ARVIND, AND RUDOLPH, L. 1999. CACHET: an adaptive cache coherence protocol for distributed shared-memory systems. *In ACM Proceedings Int. Conference on Supercomputing*, ACM, pp. 135-144.

SHIRAZI, N. AND LUK, W. AND CHEUNG, P. Y. K, 2000. Framework and tools for run-time reconfigurable designs. *In IEE Proceedings Computers and Digital Techniques 147(3)*: pp. 147-152.

STEIGER, CHRISTOPH AND HERBERT WALDER AND MARCO PLATZNER, 2004. Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-time Tasks. *IEEE Transactions on Computers*. 53(11): pp. 1392-1407, November 2004.

SWARTZTRAUBER, PAUL N. 1987. Multiprocessor FFTs; Parallel Computing, 5, 1987, pp. 197-210, 1987

VERGARA, M. AND STRUM, M. AND EBERLE, W. AND GYSELINCKX, B, 1998. A 195KFFT/s (256-points) high performance FFT/IFFT processor for OFDM applications. *In Proceedings Int. Telecommunications Symposium (ITS'98).* SBT/IEEE CS, Vol. 1, pp. 273-278.