PRADA: a high-performance reconfigurable parallel architecture based on the dataflow model

Edson P. Ferlin

Department of Computer Engineering,
Positivo University,
Rua Prof. Pedro V.P. de Souza, 5300 81280-330 Curitiba (PR), Brazil
E-mail: ferlin@up.edu.br

Heitor S. Lopes* and Carlos R. Erig Lima

Department of Electronics,
Federal University of Technology – Paraná,
Av. 7 de setembro, 3165 80230-901 Curitiba (PR), Brazil
E-mail: hslopes@utfpr.edu.br E-mail: erig@utfpr.edu.br
*Corresponding author

Maurício Perretto

Department of Computer Engineering,
Positivo University,
Rua Prof. Pedro V.P. de Souza, 5300 81280-330 Curitiba (PR), Brazil
E-mail: perretto@up.edu.br

Abstract: This work proposes and implements a reconfigurable parallel architecture based on dataflow for numerical computation, named PRADA. This architecture uses concepts of parallel processing to obtain a scalable performance and the dataflow concept for controlling the parallel execution of instructions. PRADA is composed by a control unit and several processing elements (PEs). In the control unit, there are several functional blocks, including data and instruction memories. Each PE is composed by an ALU and buffers. PRADA is organised as a cluster, in which several independent dataflow modules are interconnected together. PRADA was developed in VHDL and implemented in reconfigurable hardware using a FPGA device. Therefore, it can offer high performance, scalability and customised solutions for engineering problems. Results of the application of PRADA to the computation of a digital filter and a cryptography algorithm are presented. Results are also compared with other different architectures, such as microprocessors, ASICs, DSPs, reconfigurable architectures and dataflow architectures. In most cases, PRADA achieved either competitive or higher performance than the other architectures, regarding the measures used. Overall results suggest that this architecture can be applied to several classes of problems that may require a high throughput, such as cryptography, optimisation and scientific computing.

Keywords: parallel architecture; reconfigurable computing; dataflow.

Reference to this paper should be made as follows: Ferlin, E.P., Lopes, H.S., Erig Lima, C.R. and Perretto, M. (2011) 'PRADA: a high-performance reconfigurable parallel architecture based on the dataflow model', *Int. J. High Performance Systems Architecture*, Vol. 3, No. 1, pp.41–55.

Biographical notes: Edson Pedro Ferlin is with the Computer Engineering Department of Positivo University, Curitiba, Brazil.

Maurício Perretto is with the Computer Engineering Department of Positivo University, Curitiba, Brazil.

Heitor Silvério Lopes is with the Electronics Department of Federal University of Technology Paraná, Curitiba, Brazil.

Carlos Raimundo Erig Lima is with the Electronics Department of Federal University of Technology Paraná, Curitiba, Brazil.

Copyright © 2011 Inderscience Enterprises Ltd.

1 Introduction

Parallel computing is presented as a promising option to face the growing demand for computational power of scientific computing. For instance, computational systems named petascale have thousands of processing elements (PEs) capable of performing 1×10^{15} operations per second (Bell et al., 2006). Those architectures exploit parallel computing, aiming at reducing the overall processing time of a given task, by executing many concurrent operations (Hennessy and Patterson, 2006).

Parallel computer architectures known as dataflow were first proposed by the end of the 1970s. These architectures exploit naturally the intrinsic parallelism of the instructions of a program (Arvind and Nikhil, 1990; Šilc et al., 1999; Veen, 1986). These architectures are characterised by having a single memory for both, data and instructions, no program counter and no program variables. Values of variables are treated as packages that are transmitted between PEs. Associated to each processor, there is a template, which has all the necessary information for that processor to its job. A dataflow program is organised as a directed graph, where nodes represent instructions and edges represent the dataflow between nodes. A given node is activated, then the instruction is executed, as soon as the input data to that node is available.

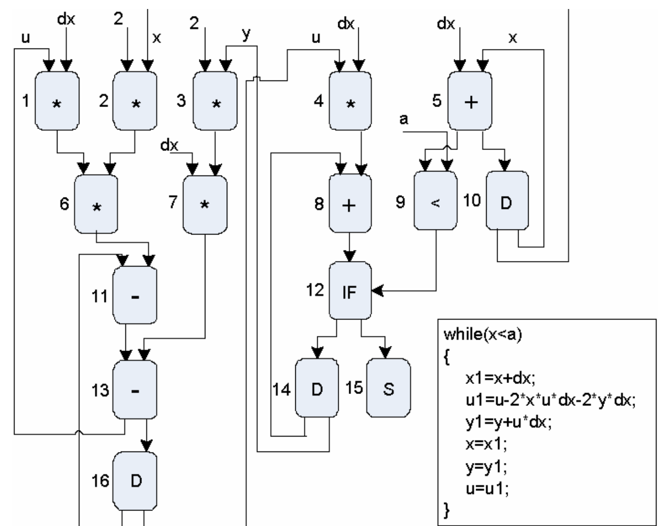
Using the dataflow model, the possible parallelism among instructions can be conveniently accessed. For instance, in the computation of the differential equation shown in equation (1), several operations can be done in parallel.

$$\frac{d^2y}{dx^2} + 2x \cdot \frac{dy}{dx} + 2y = 0, \quad y(0) = 1, y' = 0 \tag{1}$$

The numerical solution to equation (1) has 16 operations: 6 multiplication, 2 sum , 2 subtraction, 3 duplication ‘D’, 1 conditional ‘IF’, 1 relational ‘<’, and 1 stop ‘S’. Figure 1 shows the dataflow graph for the computation of this differential equation, and the corresponding C-like program. Initially, there are five independent nodes that can be executed in parallel. Next, other five, and so on, following the dataflow graph.

Reconfigurable computing is a computational model developed to join the performance of hardware-based solutions with the flexibility of software-based solutions, by using programmable devices (Becker and Hartenstein, 2003). The central idea of reconfigurable computing came up in the 1960s. However, it was materialised only after the 1980s, when powerful reconfigurable devices became available, such as the field-programmable gate arrays (FPGAs). Reconfigurable computer architecture gives great flexibility to the system, since it fits the hardware to the application, allowing the exploration of different strategies according to the task to be executed (Lysaght and Rosenstiel, 2005). In this work is used the definition of reconfigurability based on Adario et al. (1999).

Figure 1 Example of a dataflow graph (see online version for colours)



This work reports the development of a new architecture, especially devised for scientific computing, named parallel reconfigurable architecture using dataflow (PRADA) (Ferlin, 2008). This architecture is based on the dataflow model, in which the sequence of execution of operations is determined by the availability of data for processing. Another relevant feature of the architecture is that it explores the implicit parallelism of the applications and is it is fully implemented in reconfigurable hardware.

2 Related work

In the recent literature, several dataflow-based parallel architectures are reported. Amongst them, the following can be cited:

The functional computer (Quenot et al., 1993) is a mixed architecture, based on transputers and PEs known as dataflow processors. Such architecture was created for specific applications, such as real-time image processing. Besides using an outdated technological component, it has restricted applicability.

The KressArray (Hartenstein and Kress, 1995) has a matrix architecture in which the PEs are called datapath units (DPUs). The control is centralised and each DPU is composed by a floating-point arithmetic and logic unit (ALU). Operations can be done with a single data type, thus imposing limitations in the type of applications that can be run in this architecture.

The COLT (Bittner et al., 1996) architecture has a matrix of PEs named interconnected functional units (IFUs) that operate as stages of a pipeline. In order to implement the desired functions, the matrix of PEs has to be configured using a crossbar commuter. This is the main drawback of the architecture because it demands a large amount of resources.

WASMII and HOSMII (Shibata et al., 1998) are virtual data-driven hardware systems based on multiple configuration sets. The nodes and edges of a dataflow graph are directly represented in a specific FPGA configuration,

like a specific application. Consequently, this method of configuration limits the flexibility of the architecture and the number of possible configurations for a given device.

The WaveScalar (Swanson et al., 2003) is a matrix structure of PEs called WaveCache, organised as clusters. Operations are mapped into these clusters and this makes difficult the overall distribution of tasks. Additionally, the control is decentralised, causing a large consumption of resources of the programmable device for the control logic.

The asynchronous dataflow (Teifel and Manohar, 2004) is an architecture based on logic blocks that can be coupled together following a dataflow to accomplish the desired function. The standardised logic blocks operate asynchronously and the communication between them takes place by means of specific message-passing channels. The main drawback of this architecture is the several logic blocks have to be configured by means of the communication channels to implement the desired processing. This fact affects the versatility of the architecture, besides having a strong impact in the amount of resources spent.

Liu and Furber (2005) proposed a coprocessor architecture to work together with a common RISC processor. This architecture implements tasks using the dataflow model, but depends on the regular processor for general processing.

The extreme processing plataform (XPP) (PACT Technologies Inc., 2006) is a commercial architecture from PACT XPP Technologies (Los Gatos, CA, USA) that operates in a matrix structure of PEs named Processing array elements (PAE). The PAEs are configured and explicitly mapped to express the desired processing task. This feature limits the flexibility of the system, because it has to be reconfigured to be used in other subsequent tasks.

Architectures known as coarse-grained reconfigurable architecture (CGRA) usually consist of a matrix structure with a large number of functional elements, interconnected as a mesh (Galanis et al., 2007; Park et al., 2008). The reconfigurability in such architectures is at the functional level, thus assuring a flexibility and low reconfiguration overhead. Some examples of these architectures are: MorphoSys (Singh et al., 2000), PipeRench (Goldstein et al., 2000), XiRisc (Lodi et al., 2003), programmable hardware cellular automaton (PHCA) (Charbouillot et al., 2008) and DART (Pillement et al., 2008).

Additionally, other class of architectures can be also cited: those known as polymorphous computing architectures (PCAs). These architectures are mainly developed for embedded systems and can be configured either dynamically or statically to meet the computational requirements of a given application (Cavin et al., 2008; Hardnett et al., 2003). In this category it is also included the architecture TRIPS (Sankaralingam et al., 2003) and MONARCH (Vahey et al., 2006).

3 The proposed architecture

The PRADA architecture presents four basic features

1 it allows the parallel execution of algorithms

2 it allows several PEs working in parallel with a central controller

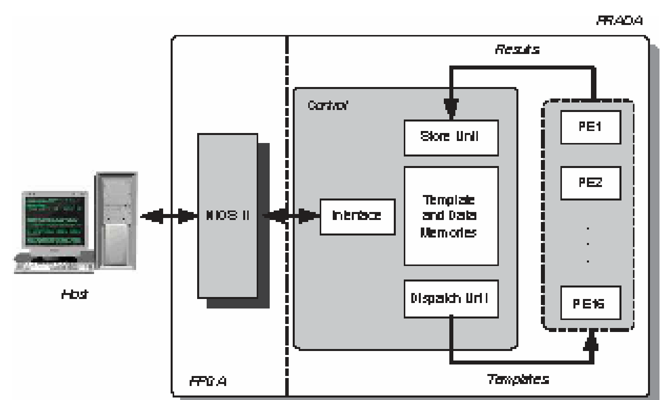
3 the control is based in the specific dataflow of the application

4 it can be semi-statically adapted to other applications.

PRADA architecture explores spatial parallelism, in which concurrency takes place due to the independency of operations, and instruction-level parallelism (ILP), where parallelism occurs as function of the instructions. However, achieving parallelism here is more complex, and requests a specialised methodology. The PRADA architecture is based on the dataflow concept. Therefore, there is no program counter, such as in other von Neumann architectures of ordinary processors, since the order of execution of instructions is determined only by the availability of the templates. There is no supervisory control over the sequence of execution besides the own template. This is important, since each template can be processed independently of the other, if the necessary data and free PEs are available for processing.

The PRADA architecture is composed by a single control block and several PEs, as shown in Figure 2. Therefore, considering the control system, the proposed architecture is different from WaveScalar (Swanson et al., 2003) that has a decentralised control, and similar to KressArray (Hartenstein and Kress, 1995), i.e., the control is also centralised. The control block is responsible for managing the architecture and communication, receiving data and sending results. This block also manages the transmission of templates (operations) to the PEs, by means of parallel buses. It also is responsible for the storage of processing.

Figure 2 PRADA architecture (see online version for colours)



The PRADA system is connected to a desktop computer used as host that is responsible for providing an interface for the user. The communication between PRADA and the host is done by means of a Universal Serial Bus (USB) interface and software for transmitting templates and receiving results. USB is a standard interface that al lows the connection with many systems, it is easy to implement and facilitates the reproduction of the work. An interesting

alternative would be the PCI/PCIex interface that would improve significantly the communication between host and the architecture. However, the development kit available for this work is external and do not provide a PCI/PCIe connection. This alternative interfacing will be explored in future work. In the FPGA, the communication is done using the NIOS II embedded processor. Besides providing an interface to the user, the host also is responsible for sending templates to PRADA and receiving back the results. In the host, there is also the development environment as well as the software for manually generating the templates, where the user is requested to fill all fields of the templates. Complex applications may require the use of programming languages that are compiled to dataflow graphs (Šilc et al., 1999), although manual construction of the templates, when possible, allows a deeper insight in the process.

As mentioned before, the template corresponds to the instructions each PE executes. Each template has all the information necessary for its execution and it is organised in five fields: Operation, Operand 1, Operand 2, Destination 1 and Destination 2, as shown in Figure 3. The field operation has the opcode that identifies the logical operation to be done with operands. Operand 1 and 2 store the data that will be manipulated by the PE. Fields Destination 1 and Destination 2 have the information to where the result of the operation will be sent. The template width is 66 bits because, besides the fields mentioned, it also have ten extra bits used for internal control (origin of data, validity of operand, destination port, and other). It is worth to mention that, in this model, there are no subsequent readings from memory to operands, as in the von Neumann model.

Figure 3 Basic structure of a template

Opcode	Operand 1	Operand 2	Destination 1	Destination 2	Control
4 *bits*	16 *bits*	16 *bits*	10 *bits*	10 *bits*	10 *bits*

The execution cycle of a program is done by the control block of Figure 2 following the application dataflow. A more detailed view of the control block is provided in Figure 4, and its five main components are:

- *Interface*: Provides the communication channel with the host by means of an embedded processor using memory buffer and an USB interface.
- *Template memory (TM):* This is where the templates are stored for further processing. From the logical point of view, the TM is a single memory that can store the maximum amount of templates. Physically, this memory is divided into several separated modules, allowing other modules to access the templates in parallel. The memory segmentation method is based on the principle of locality: references to the memory in a given time interval tend to access a specific region of memory (Tanenbaum, 2005). Therefore, the probability of consecutive templates being executed simultaneously is high or, at least, reasonably higher than other cases. The TM is capable of storing up to 1 K templates, 66 bits-long. It is segmented in four parts of the same size, in which positions are interleaved.
- *Data memory (DM):* This is where data are stored, basically, data structures such as vectors and matrices. It is structured as 1 K words, 16 bits-long. There is a single DM for the whole system and PEs are not allowed to access it, since they receive the necessary data already incorporated in the templates.
- *Dispatch unit (DU):* This block continuously verifies if there are templates to be executed and send them to the available PEs. The DU is composed by three blocks: fetch – responsible for seeking for templates in the TM and data in the DM, and ensemble the templates to be processed; buffer – stores the templates ready to be dispatched to PEs; dispatch – effectively dispatches templates for PEs.
- *Storage unit (SU):* This block receives the processing results from PEs and updates the appropriate templates in the DM. It is composed by three blocks: store – controls the update of templates with data resulting from processing; snoop – responsible for updating the status of templates when DM is updated; update – receives the new requests from DM. It preserves the consistence of memory assuring that data will be updated in the DM.

The PEs, shown at the top of Figure 4, are the core of PRADA. The PEs are identical and they are composed by ALUs capable of performing addition, subtraction, multiplication, division, logical operations, relational and conditional operations. This feature is different from most of the previously mentioned architectures, such as: XPP, WaveScalar, KressArray, WASMII, HOSMIII and COLT, in which the operations are mapped directly into PEs. Besides, each PE also has its own set of input and output buffers to facilitate synchronisation with the associated DU and SU. The structure of a PE allows the execution of the above mentioned operations with two operands (monadic or dyadic) and produces as output a single result that can be forwarded to two destinations. In PRADA, 16 PEs were implemented, all identical and grouped into four sets, i.e., a 4×4 processing cluster. The cluster structure was used in PRADA in order to minimise the bottleneck for accessing the TM, in such a way that even if a cluster is sequentially accessing a TM, the remaining clusters can operate in parallel the remaining TM modules.

The logical structure of PRADA recalls a multiprocessor cluster with four nodes, each one with four processors. There is also a shared DM that is not directly accessed by the PEs. Each node of the cluster is composed by four PEs, two SUs, one DU and one TM. The whole ensemble forms a dataflow machine with four functional units. In Figure 5 the logical structure of the parallel architecture of PRADA is represented, detailing a node of the cluster. In this configuration, there is a homogeneous parallel environment with two memory access models (Pfister, 1998): uniform memory access (UMA) for the DM; and non-uniform

memory access (NUMA) for the TM. Such duality is aimed at taking advantage of the best for both cases. The use of the UMA model guarantees the coherence of data in the DM. On the other hand, the use of the NUMA model for TM minimises the bottleneck of memory access, since each cluster (DF – dataflow) accesses directly one TM module.

Figure 4 Detailed view of the physical organisation of PRADA

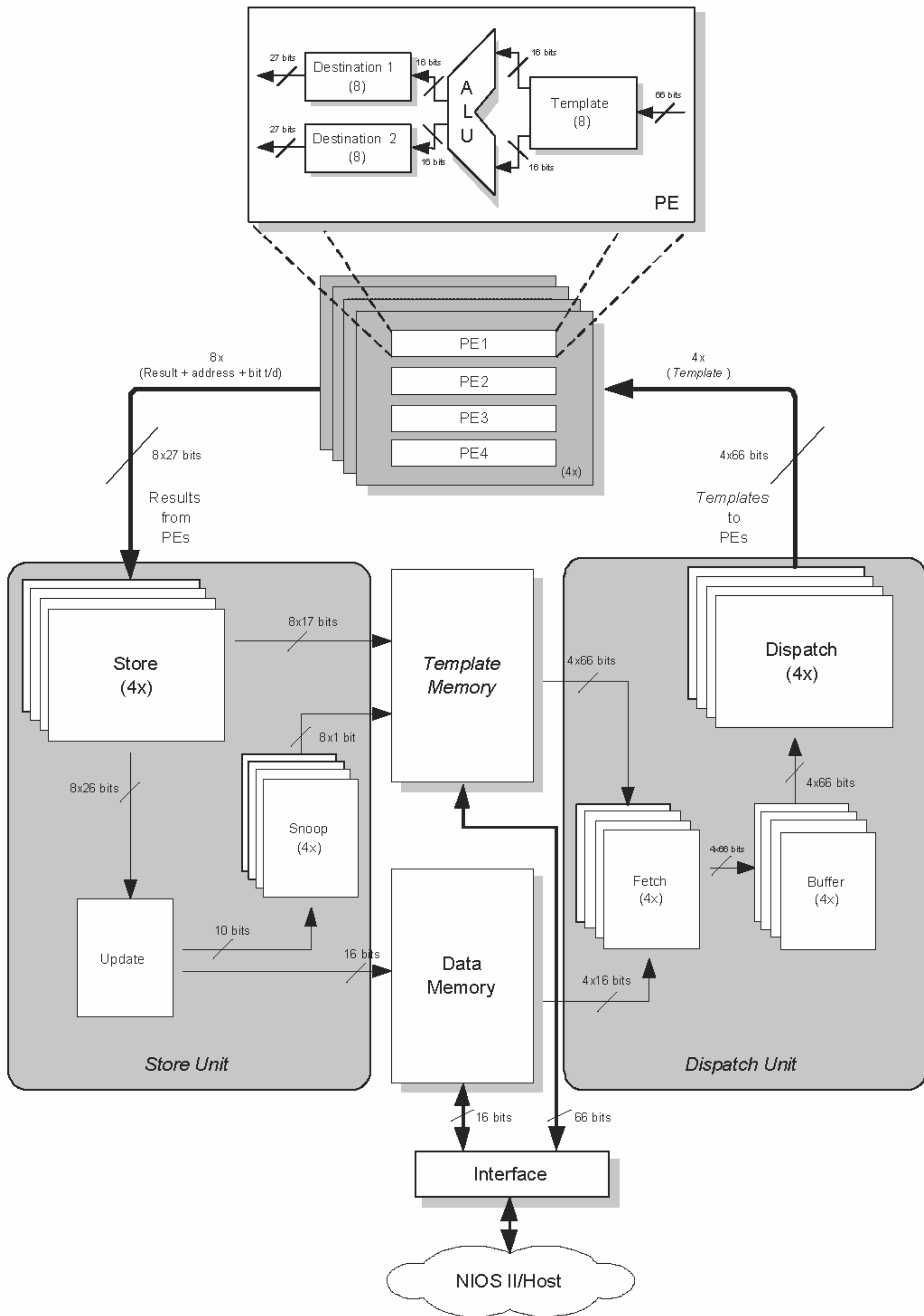
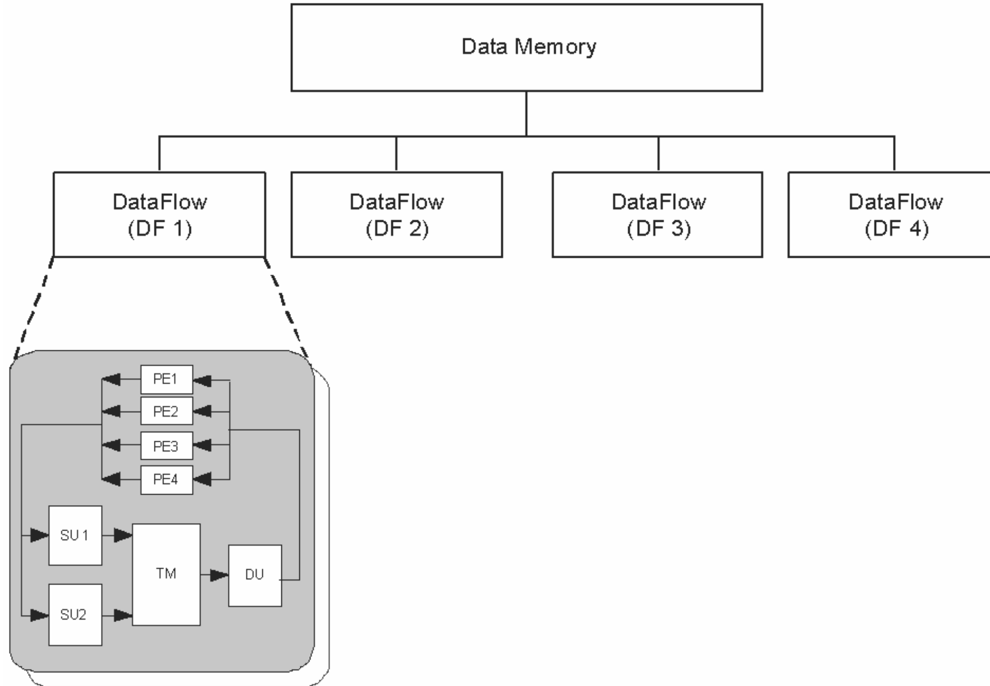


Figure 5 Logical structure of PRADA



The processing cycle of PRADA architecture is detailed algorithmically in Figure 6, and explained as follows. Initially, the host sends to PRADA the program to be executed in the form of a set of templates, which are stored in the TM. It also sends all data structures to be loaded in DM. A signal from host starts processing. After, the DU enquires the TM and forwards to the PEs the templates to be executed, as soon as they become available. The PEs, once receiving, perform the operation described by each template and produce results that are stored in TM by SU. This process is repeated until the TM is empty.

Figure 6 Processing cycle in the PRADA architecture

1: Load TM with templates from host;
2: Load DM with data from host;
3: Wait for start;
4: Start in parallel all the functional blocks:
  a) DU_Fetch - seeks templates and data;
  b) DU_Dispatch - sends templates to PEs;
  c) PEs - execute templates;
  d) SU_Store - stores operands/data in TM and DM;
  e) SU_Snoop - checks for templates to be dispatched as consequence of data availability in DM;
5: Send DM contents to host;
6: Send TM contents to host.

The scheduling of operations is automatically done by PRADA. This is due to the central idea of executing an instruction (template) as soon as the necessary data is available and there is an idle PE to process such instruction.

4 Implementation

PRADA was developed to be implemented without the need of any other additional component (for instance, microprocessors, external memories, etc), as in other architectures [for instance, in Liu and Furber (2005), Quenot et al. (1993)]. This feature makes PRADA as device-independent as possible, avoiding to be technologically outdated.

PRADA was implemented using reconfigurable logic, using high-density FPGA devices. By using reconfigurability the logic blocks of the FPGA are reconstructed to implement the necessary logical functions. The reconfiguration of this architecture is semi-static. That is, it occurs immediately before the execution of the application.

The physical implementation was done in a FPGA device of the Stratix II family from Altera (http://www.altera.com), namely, the EP2S60F672C3 device. The implementation reported in this paper was done with 16 PEs. The amount of resources required for such implementation was: 28,348 logic elements and 132,160 memory bits, corresponding, respectively, to 59% and 5% of the available resources of the device. Each PE needs 535 logic elements – less than 2% of the device. We used the Quartus II development system from Altera and all blocks were developed in standard very-high-speed-integrated-circuit hardware description language (VHDL) (Pedroni, 2004).

The implementation runs at 50 MHz and, with this 20ns clock cycle it is possible to execute 13×10^6 operations per second using a single PE. Using 16 PEs, the maximum attainable performance is approximately 65 MIPS. This is obtained considering a best-case algorithm, in which all instructions are independent.

PRADA has the following features showing in Table 1. These features can be modified for meeting the requirements of a given application, by means of reconfigurability. Such parameters can be changed in the project of the architecture, before running the application, by sending a new configuration to the device.

Table 1 Features of the PRADA architecture

Feature	*Value*
Number of PEs	16
Operating frequency	50 MHz
DU -PE bus width	4 × 66 bits
DU -PE bus bandwidth	13.2 Gbps
PE -SU bus width	8 × 27 bits
PE -SU bus bandwidth	10.8 Gbps
Number of instructions	16
Data size	16 bits (integer)
Maximum number of templates	1 K
DM	1 K words (16 bits)
TM	1 K long word (66 bits)
Template size	66 bits

The number of PEs and the operating frequency depend on the constructive limitations of the device used in the implementation. These features of PRADA are a direct function of the technological advance of FPGA devices. On the other hand, the width of the buses and data, the size of internal memories and the number of instructions are defined by the design of the architecture.

The PRADA architecture has a superscalar structure (Tanenbaum, 2005), basically, with a three-stage pipeline (dispatch, processing and storage) that divides the execution of instructions in several parts, shown in Figure 7. These parts are executed in parallel, each one being processed by a dedicated hardware with a specific function. A superscalar structure requests the use of a dedicated pipeline for each functional structure in the architecture (Stallings, 2009). Due to the pipeline, there is an overlap in the execution time of the instructions within the several units, in such a way that all these units will explore time parallelism. Therefore, pipelining is an important intrinsic feature of the architecture that leads to the improvement of the overall performance. This instruction overlap is known as ILP, and corresponds to the lowest possible level of parallelism that can be explored in computer architecture (Hennessy and Patterson, 2006).

The instructions set of PRADA has 16 basic instructions, grouped as: arithmetic (sum, subtraction, multiplication and division), logical (AND, OR, NOT, XOR, NOR), conditional (If), relational (=, <>, >=, <) and miscellaneous (duplication, stop). Such instructions set were chosen taking into consideration the nature of the numerical processing of scientific applications, the primary focus of the PRADA architecture. The instruction set is summarised in Table 2. The execution time for most of instructions is 11 clock cycles, with exception to division and conditional that takes 12 clock cycles. In the case of instructions that have two destinations, an extra clock is necessary for the additional destination. The STOP instruction is executed automatically by DU and takes only one clock cycle, the time necessary for finishing the fetch for new templates.

Figure 7 Pipeline associated with the PRADA architecture

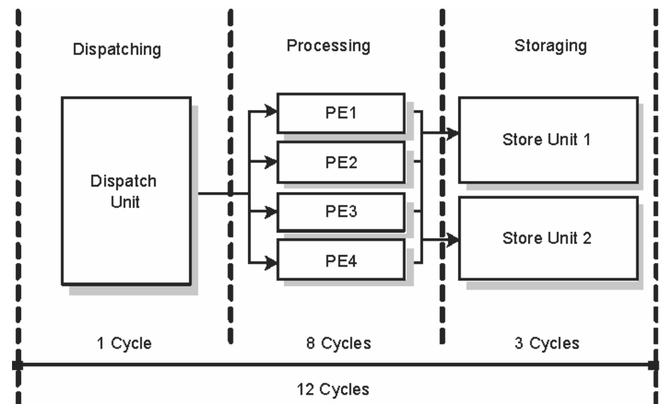


Table 2 Instruction set of PRADA

Mnemonic	*Operation*	*Opcode*	*#Clock*
ADD	Sum	0000	11
SUB	Subtraction	0001	11
MUL	Multiplication	0010	11
DIV	Division	0011	12
AND	Logical AND	0100	11
OR	Logical OR	0101	11
NOT	Logical NOT	0110	11
XOR	Logical EXCLUSIVE-OR	0111	11
NOR	Logical NOT-OR	1000	11
CEQ	Compare equal	1001	11
CNE	Compare not equal	1010	11
CGE	Compare greater or equal	1011	11
CLT	Compare less than	1100	11
IF	Conditional	1101	12
DUP	Duplication	1110	11
STOP	Stop	1111	1

A simple example of dataflow program that can be run in PRADA is shown in Table 3 for the numerical solution of the ordinary differential equation [equation (1)], subject to $y(0) = 1$ and $y'(0) = 0$. This dataflow program, shown as pseudo-templates, corresponds to the dataflow graph of Figure 1. In Table 3, Op. 1, Op. 2, Dest. 1 and Dest. 2, mean, respectively, the two operands and the two destinations of the operations' result.

Table 3 Templates of a dataflow program corresponding to the graph of Figure 1

Node	*Mnemonic*	*Op. 1*	*Op. 2*	*Dest. 1*	*Dest. 2*
1	MUL	*u*	*dx*	6	0
2	MUL	2	*x*	6	0
3	MUL	2	*y*	7	0
4	MUL	*u*	*dx*	8	0
5	ADD	*dx*	*y*	9	10
6	MUL	0	0	11	0
7	MUL	*dx*	0	13	0
8	ADD	0	0	12	0
9	CLT	0	*a*	12	0
10	DUP	0	0	2	5
11	SUB	0	0	13	0
12	IF	0	0	14	15
13	SUB	0	0	1	16
14	DUP	0	0	8	9
15	STOP	0	0	0	0
16	DUP	0	0	11	4

5 Computational experiments

For testing the performance of PRADA, three problems were used: best-case (BC), finite impulse response (FIR), and international data encryption algorithm (IDEA).

The first is a toy problem composed of 192 independent operations with 16 bits words and fixed-point arithmetics. The objective of this problem is to allow PRADA to demonstrate its maximum performance in an idealised situation, where there is no data dependence between templates.

The second problem is a scientific computation. It is related to the computation of a five-tap digital filter. This is real-world problem in which the application imposes limitations to the performance of the architecture due to the data dependency between operations. In this problem, in the same way, we used fixed-point arithmetic with 16 bits resolution. The main reason for choosing this problem is that it has been used as benchmark for testing several architectures reported in the literature. Therefore, it can be used as a basis for comparison. A FIR digital filter is characterised by a response to the unity impulse that becomes null after a finite time. Figure 8 presents the algorithm for computing the FIR filter, written in C programming language.

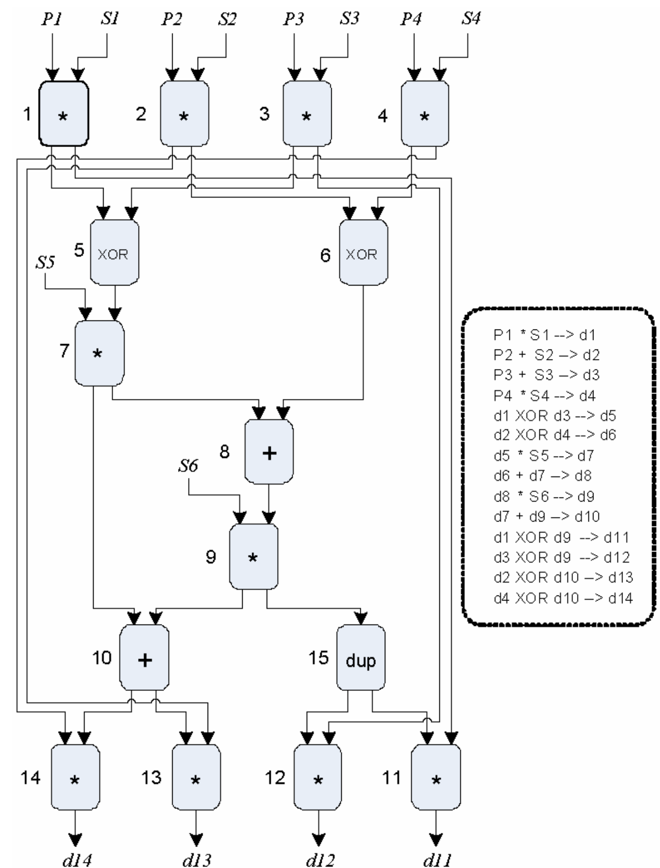
Figure 8 Algorithm for computing a five-tap FIR filter, in C language

```
for (i=0; i<62; i=i+1)
{
  y[i]=h0*x[i] + h1* x [i+1] + h2* x [i+2] + h3* x [i+3] + h4* x [i+4];
}
```

The IDEA is a cryptography algorithm created in the 1990s (Lai and Massey, 1990). The algorithm operates on 64 bits-wide in-formation blocks, using a key of 128 bits. IDEA is considered to be immune to differential cryptanalysis, and there is no known algebraic weakness in it (Schneier, 1996). Although IDEA has been patented in a number of countries, it is freely available for non-commercial use. This algorithm was chosen for experiments with PRADA not only due to its simplicity, but also due to the fact that some operations can be done in parallel. Basically the algorithm uses three operations: bitwise exclusive-OR, modulo 216 addition, and modulo 2^{16} + 1 multiplication. Both encryption and decryption procedures are similar. A total of eight identical transformations, plus an output transformation, are done.

Figure 9 shows the dataflow graph of the IDEA. It is possible to observe that some operations can be executed in parallel (at most, four), and some must be done sequentially, a constraint inherent to the algorithm. In this algorithm, variables *P*1 to *P*4 represent each one a 16-bit block of information, which, together, yields a 64-bit ciphertext. Variables *S*1 to *S*6 are 16-bits cryptography sub-keys. The encryption process need eight iterations of the algorithm shown, using a total of 52 sub-keys generated from a key of 128 bits.

Figure 9 Dataflow graph for the IDEA algorithm (see online version for colours)



For all problems, the pseudo-templates were manually constructed, based on the algorithm and/or dataflow graph for the applications.

6 Results

In the following sections, the results of the experiments for the above mentioned problems are reported and grouped according to the nature of the experiments:

1. Scalability experiments (Section 6.1), in which the objective is to verify the behaviour of PRADA, based on the maximum attainable performance with an idealised problem and a real-world problem, in comparison with the ideal response as function of the number of PEs.
2. Comparison experiments (Section 6.2), in which the main objective is to compare the performance of PRADA with other architectures, some of them somewhat similar and other quite different. Such comparison was done based on experiments and reported results found in the literature, using the FIR and IDEA problems.

6.1 Scalability experiments

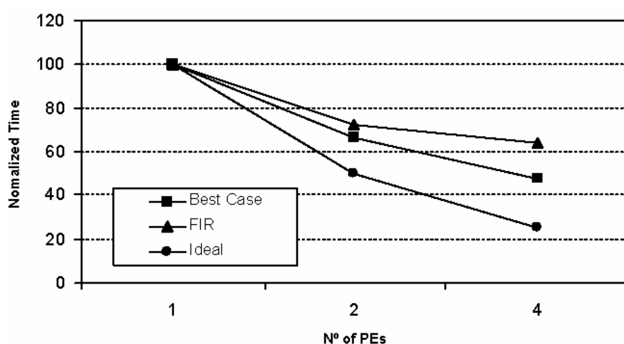
Experiments of this category were done using PRADA in two configurations:

a. with 4 PEs and a single TM module
b. with PEs and the TM segmented in four modules.

6.1.1 PRADA with four PEs and a single TM module

Experiments done with BC and FIR algorithms, using a single memory (like a single DF module), considering one, two and four PEs, indicate that the execution time were above the ideal processing time. The ideal processing time is obtained by dividing the number of machine cycles spent with a single PE (that is, sequential processing), by the amount of PEs. For instance, using four PEs, the ideal processing time should be 1/4 the time of a single PE. The difference of time between the ideal and the time taken by BC algorithm is 36.41%. Similarly, the difference between the ideal time and the FIR algorithm is 45.85%, as shown in the graph of Figure 10. The difference of 9.44% between the execution time is mainly due to the data dependencies inherent to the FIR algorithm presented before.

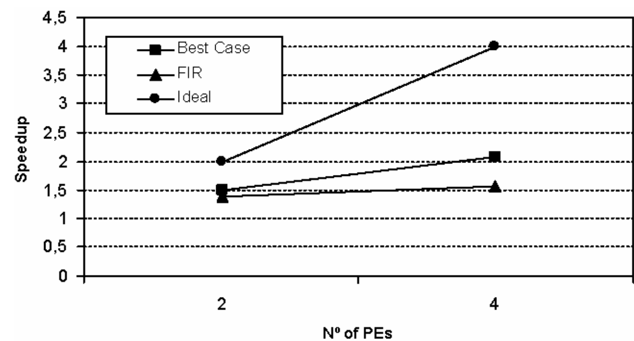
Figure 10 Execution time for BC and FIR algorithms, using a single memory



The curve corresponding to the BC represents the best possible result that can be attained by the execution of an application using PRADA. The ideal curve represents idealised hypothetical values when considering that the architecture does not present any limitation and the application takes the maximum advantage of available resources.

A usual measure of performance in parallel architectures is speedup (Murdocca and Heuring, 2007), that is, the ratio between the execution time with a single PE and the execution time with two or more PEs in parallel. For the BC algorithm, the speedup achieved 63.59% of the ideal value, running with two and four PEs. Similarly, for the FIR algorithm, the speedup achieved 54.15%, as shown in Figure 11. For both cases, PRADA was run using a single memory (TM) module. Therefore, there is coherence in the access to the TM, causing the system to spend extra time due to the serialisation of the access. Obviously, this is a problem that negatively affects the overall performance of the system. For this specific analysis, we consider that the ideal value for speedup is the number of PEs in the architecture, for instance, using two PEs, the ideal speedup would be 2.

Figure 11 Speedup curves for BC and FIR algorithms, using single memory

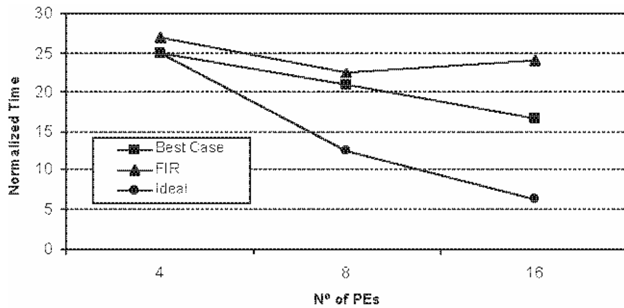


6.1.2 PRADA with 16 PEs and TM segmented in four modules

The same experiments reported before were done, reconfiguring PRADA for using the TM divided into four modules, using 4, 8 and 16 PEs. Results, regarding processing time, are shown in Figure 12. It can be observed that, using four PEs, the processing time for the BC algorithm was the same as the ideal processing time. This is due to the optimisation of the instructions, since the templates were constructed in such a way to obtain the maximum possible parallelism of operations. On the other hand, the average processing time for BC algorithm was 34.17% above the ideal time. For the FIR algorithm, PRADA achieved a processing time 41.84% above the ideal time. The difference of execution time between algorithms BC and FIR was 7.67%, except when using 16 PEs, when the difference was 11.58%. This difference represents the percentage of serial part of the FIR algorithm, considering that the BC algorithm is the performance reference that can be achievable in the architecture. This fact indicates a

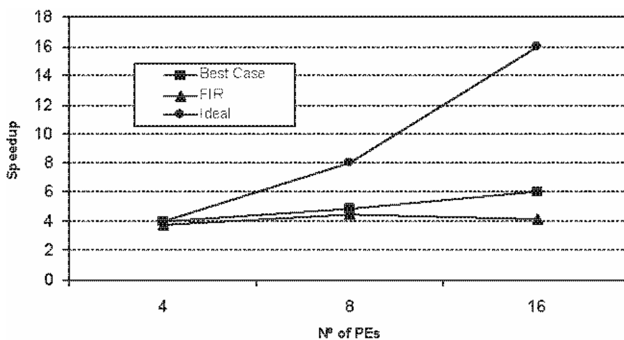
limitation of the architecture when running the FIR algorithm and it is due to the many memory accesses caused by the data dependencies of the application.

Figure 12 Running time for BC and FIR algorithms with segmented memory



Regarding the speedup, PRADA with TM segmented into four modules achieved an average of 49% of the ideal value for the BC algorithm, except when using four PEs, because the speedup obtained is equal to the ideal. These results are shown in Figure 13. The same experiment with the FIR algorithm showed that PRADA achieved 74.28% of the ideal value, in average. Again, when using four PEs, the speedup was very close to the ideal value.

Figure 13 Speedup curves for BC and FIR algorithms using segmented memory



The PRADA architecture is scalable, since we can use a larger number of PEs for both models (single and segmented memory), other than those presented in this work. However, we used a configuration that included a maximum of 16 PEs, aimed only to demonstrate the validity of the underlying concepts, not for achieving the maximum performance. Consequently, it is possible to use a larger number of PEs, but with some modifications in the architecture. On the other hand, it is possible to achieve higher scalability using the concept of cluster, in which each node is a PRADA machine.

6.2 Comparison experiments

Another set of experiments was done aimed at comparing PRADA with other architectures found in the literature. The first is the FIR filter mentioned before in Section 5 and the second is the IDEA algorithm for cryptography. These two problems were chosen due to the availability of comparison data in the literature.

6.2.1 FIR algorithm

Table 4 shows several important information for comparing performance of different architectures: operating frequency, in MHz (Freq.); execution time, in clock cycles (#Clock); number of instructions executed (#Instr.); throughput; CPI and MIPS. Throughput, in this case, is defined as the ratio between the number of operations of the tap by the number of clock cycles. In this case, the throughput represents a rate of the overall processing that is executed taking into consideration a time interval of a single clock pulse. CPI (Clock Per Instruction) is defined as the average number of clock cycles spent per instruction executed, for a given problem. IPC, on the other hand, is the inverse of CPI. IPC is used to compare architectures considering the number of instructions executed each PE. MIPS (Millions of Instructions per Second) is a common measure of computer speed, and it is a measure strongly influenced by the application. Consequently, it can be used with care, since different values can be obtained for the same architecture, but running different applications. In that table, we also provide the ratio IPC per processing element (IPC/PE) and number of clock cycles necessary per filter tap (cycles/tap).

For this experiment, PRADA was configured with a single PE, and was compared it with a general-purpose desktop computer (PC). The PC used for comparison had a AMD Athlon XP64 3000+ processor, running at 2.17GHz, with 512 MBytes of RAM and Microsoft Windows XP operating system and using Borland C++ Builder 6 (standard configuration without optimisation). In the PC, the algorithm took 0.358 μs (that is, 777 cycles × 1/2170 MHz), far below from the time taken by the proposed architecture with 1 PE (9.04 μs) (452 cycles × 1/50 MHz). However, it should be noted that the clock of the PC is around 43 times faster than that used in the reconfigurable architecture. A more fair comparison is considering the number of clock cycles. For this case, the PC needed 777 clock cycles and the one-PE-architecture needed only 452 clock cycles. On the other hand, the relative PRADA low clock operation is a technological limitation. New FP-GAs families will enable higher clock rates, improving the PRADA performance.

We compared the values obtained in these experiments with other similar results running in different architectures, but for the same algorithm: ROCCC – reconfigurable Computing Compiler System, OGMS – optimisation generation memory structure for window operations – see Dong et al. (2007), AMD Athlon processor in a PC, Intel 8051-microcontroller and Altera NIOS II/e embedded microprocessor. Recall that the ROCC and OGMS architectures are specifically designed for digital filtering.

PRADA, with 16 PEs needed a number of clock cycles around 2.4 times smaller than the specialised architectures, i.e., 18.2, 78.5 and 7.1 times smaller than NIOS II/e, 8051 and PC, respectively.

Table 4 Comparison of the performance of several architectures for the FIR filter

Architecture	*#PEs*	*Freq.*	*#Clock*	*#Instr.*	*Throughput*	*MIPS*	*CPI*	*IPC*	*ï¿½ PC / PE*	*Cycles/tap*
					PRADA					
1 PE	1	50	452	122	0.56	13	3.70	0.27	0.25	7
4 PEs	4	50	122	122	2.07	49	1.00	1.00	0.15	2
8 PEs	8	50	101	122	2.50	60	0.83	1.20	0.11	2
12 PEs	12	50	94	122	2.68	65	0.77	1.30	0.07	2
16 PEs	16	50	109	122	2.31	56	0.89	1.12		2
					Application specific using Xilinx FPGA					
ROCCC		94	262		0.96					4
OGMS		239	263		0.96					4
					Soft processors					
NIOS II/e	1	50	1986	841	0.13	45	2.36	0.42		32
SPREE	1	80	1145	822	0.22	53	1.36	0.74		18
					Microprocessors					
AMD Athlon (PC)	1	2170	777	2158	0.32	5935				13
8051	1	12	8559	713	0.03	1	12.00	0.08		138
					Digital signal processors (DSP)					
BF 533	1	750	186			500				3
TMS 320C2	1	300	248			12				4
TMS 320C5X	1	29	434			50				7
TMS 320C55X	1	200	504			400				4
					Reconfigurable architectures					
KressArray	24	25	558							9
COLT	16	50	496							8
Wavescalar	512						0.01	71.4	0.14	
TRIPS	256						0.09	11.00	0.04	
					Dataflow architectures					
Manchester	20	15				6	2.50	0.40	0.02	

Based on the results shown in Table 4 for the FIR algorithm, it is observed that PRADA presented better performance, regarding clock cycles, when compared with dedicated architectures (ROCCC and OGMS) (Dong et al., 2007), digital signal processors (BF 533, TMS 320C2, TMS 320C5x, TMS 320C55x) (Myjak and Delgado-Frias, 2008; Petersen, 1995; Varnagiryte et al., 2002), embedded processors [Altera's NIOS II and SPREE (Yiannacouras et al., 2007)], general-purpose processors (AMD Athlon and 8051), as well as reconfigurable architectures [KressArray with 24 PEs (Hartenstein et al., 1994), COLT with 16 PEs (Cherbaka, 1996), Wavescalar (Swanson et al., 2003) and TRIPS (Sankaralingam et al., 2003) and a Manchester dataflow architecture (Šilc et al, 1999)].

This can be also verified if considering cycles per tap that measures objectively the performance for the specific application. Regarding this measure, PRADA obtained the highest performance, thus indicating its efficiency for this kind of scientific computing. PRADA achieved, for instance, a value twice smaller than other dedicated architectures and 4.5 times smaller than reconfigurable architectures.

Notice the small number of instructions executed by PRADA to accomplish the task, when compared with the other approaches. PRADA needed only 122 instructions, while other architectures a much larger number of instructions was needed: 17.7 times more in a PC Athlon, 5.8 times more in a 8051 and 6.9 times more in the NIOS II embedded processor. It is also important to recall at this point that PRADA has a reduced set of instructions, as shown in Table 2, when compared with the other architectures.

Comparing PRADA with Wavescalar (Swanson et al., 2003), we observe that the latter achieves an IPC per PE of 0.14, while PRADA achieves an average value of 0.15. This represents an improvement of around 8.5%, meaning that PRADA is more efficient than Wavescalar regarding the real parallelism that takes place in the execution of

instructions, measured as the number of instructions executed per clock cycle. The very same analysis is true when comparing PRADA with TRIPS (Sankaralingam et al., 2003), in which the difference is 3.75 times higher, since the IPC of TRIPS is 0.04.

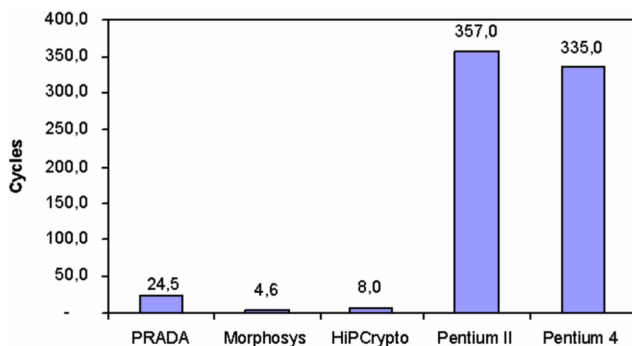
The simulation of the FIR algorithm was done in the Simulator software of Quartus-II (Altera Inc.) environment. For this experiment, PRADA was configured with 16 PEs and memory split in 4 modules. The overall processing time took 2.18 μs, that is, 109 clock cycles (20ns each), resulting in a speedup of 4.15 relative to a sequential execution.

6.2.2 IDEA algorithm

The implementation of PRADA architecture using 16 PEs was done with a EP2S60F672C3 device from Altera, and needed 28,348 logic elements and 132,160 bits of memory. We also implemented IDEA in a software version, in C++, to run in two PC's, one with a Pentium II processor and other with a Pentium IV processor. We compared results found in the literature for other dedicated architectures, namely, Morphosys (Singh et al., 2000) and HiPCrypto (Salomão et al., 1998). Morphosys is an architecture with single instruction multiple data (SIMD) coprocessors, in which operations are mapped into PEs. It was specifically adapted for running IDEA. HiPCrypto is a specialised hardware implementation for IDEA, based on ASIC.

PRADA produced ten ciphertext blocks in 245 cycles, although this number could be reduced if more were PEs were used in parallel. The only limitation is the size of the TM, necessary to store parts of templates of the algorithm. Only for comparison purposes, authors of Morphosys and HiPCrypto report that the first needed 73 cycles to produce 16 ciphertext blocks, and the second produced seven in 56 cycles. To produce a single ciphertext block, the software version needed 357 and 335 cycles, respectively, when running in a Pentiun II and a Pentium IV. The overall relative amount of clock cycles per ciphertext block is represented in Figure 14. Consequently, the proposed architecture achieved a significant gain compared with the Pentium architectures, and a performance small behind Morphosys and HiPCrypto.

Figure 14 Number of cycles for one ciphertext block in different architectures (see online version for colours)



7 Conclusions

This work presented PRADA, a new parallel dataflow architecture implemented in reconfigurable logic. Several experiments were done with three algorithms: a set of instruction without data dependence (best case – BC), a scientific application (finite impulse response digital filter – FIR) and a cryptography application (IDEA). The comparison of PRADA with other architectures focused on performance. Results showed a significant reduction of processing time with PRADA. Even comparing PRADA with architectures of higher MIPS (due to higher clock frequency), we observed that PRADA is more efficient regarding the number of clock cycles necessary to carry out the FIR algorithm.

Finished the experiments, we observed that the processing time decreases as more PEs are added to the architecture (see Table 4). This expected result can be also checked in the graphics previously shown (Figures 10 to 13), where the curves for real and ideal processing time are close each other. However, the visible difference between these curves is due to limitations of the architecture, such as memory access latency time, as well as application-dependent dependencies.

Results indicate that PRADA needs a small number of operations to carry out a given task, as in the case of the FIR algorithm (only 122 operations). This value is significantly smaller when compared with other architectures, such as the PC Athlon, 8051 and NIOS II. Analysing an application-dependent metric, the number of cycles per tap (of the FIR filter algorithm), PRADA had the smallest value (2), comparing with the other architectures. This is another interesting indication of the suitability of the proposed architecture for such kind of numerical computation. Regarding IPC, the absolute value for PRADA is small, but considering the ratio IPC/PE, the value is above all other architectures, indicating that more instructions are executed per clock cycle than in the other implementations. A similar analysis of CPI can be done, but, in this case, the small, the better. Comparing the throughput the other architectures, PRADA achieved the best performance, again supporting the assertion that PRADA is suitable for numerical computation. The number of clock cycles spent by PRADA with 16 PEs is smaller than any other architecture, including those specialised in numerical computation.

Several architectures previously mentioned, namely, XPP, WaveScalar, KressArray, WASMII, HOSMII and COLT have their operations mapped in the PEs. Differently from them, PRADA has identical PEs with a predefined set of operations, selected by programming as function of the application. In the current experiments, the instructions set were restricted to only 16 basic operations. Notwithstanding, such instructions set was adequate and enough to cope with a large range of applications, especially those related to fixed-point numerical computation.

For the IDEA algorithm, we observed that PRADA was 15 times faster than the software versions running in PC. However, PRADA was slower than Morphosys (five times) and HiPCrypto (three times). It is important to recall that

these two architectures are either designed or optimised for cryptography, while PRADA is generic, not specific for this problem. This experiment showed again the flexibility of the proposed architecture to adapt to a real-world problem.

Architectures that map directly the operations into PEs are usually arranged in a matrix structure. Such models are naturally less flexible than other approaches, and demands more resources from a reconfigurable logic device. Besides, such architectures have PEs with predefined functions. Therefore, they cannot be reused in other applications which operations were not previously defined in the FPGA configuration. Depending on the way operations are mapped in the matrix structure, a large idleness of PEs may take place. As a consequence, most of architectures that use a matrix structure have a large number of PEs (see Table 4). This is because the number of PEs must be equal of larger than the number of different operations required by the application, thus establishing bounds in the applicability of the architecture.

The problems mentioned before were circumvented in PRADA, thanks to the way its architecture was devised. Recall that PRADA has identical PEs with predefined operations that can be reused during processing for different functions, without limiting the number of operations of the application.

Although not new, the concept of dataflow is a great idea that takes several advantages for computer architectures. Mainly, the scheduling of operations is naturally performed following the dataflow graph of the application, thus reducing the control logic. As a side effect, this philosophy explores the intrinsic parallelism inherent to the program.

Another important issue in this work is how parallelism was explored, not only in the operations/instructions, but also in the functional blocks of the architecture. PRADA explores naturally the functional parallelism by using several PEs. This is the most evident level of (spatial) parallelism in the proposed architecture. The pipeline structure present in PRADA enables another level of parallelism. This is a temporal parallelism, because simultaneous operations can be started even before the current operations have finished. Besides, there is also the parallelism resulting from the fact that multiple functional blocks (such as DU and SU, for instance) are independent each other and work based on events. Therefore, one can have, in a given moment, all those blocks working simultaneously. Another interesting point is that PRADA adjusts itself to the number of available PEs for the current execution, thus avoiding the need of changing the application code. This is due to the logic of the DU that directs the templates ready to run straight to the PEs, as soon they are available. Therefore, there is a relationship between the operations and the PEs, allowing the amount of PEs to be modified according to the availability of resources in the FPGA device, thus making PRADA fully scalable.

It is important to point that a parallel architecture implemented in FPGA is a viable approach over traditional hardware design: it allows a fast reconfiguration of the project to adequate to other applications, better adequacy of the hardware to the application, reuse of resources and shorter development cycle. Also, the design flexibility of reconfigurable computing allows the functional features of the architecture to be expanded so as to meet the processing requirements of the application, for instance, increasing the number of PEs. In the case of PRADA, the internal structure was limited to 16 PEs for the experiments reported here. To increase its processing capacity, it would be enough to turn to a more powerful FPGA.

PRADA was fully developed using VHDL, a standard hardware description language. Therefore, the project is platform-independent, facilitating the migration to other development environments and reconfigurable devices. However, a known drawback of this approach is the difficulty of development, when compared with a software project, for instance. Several complex issues emerge in the development of reconfigurable systems, such as the limitation of resources of the FPGA and difficulty in debugging and testing the system. Besides, there is a technological limitation regarding the clock rate. However, more recent FPGAs devices certainly will be faster.

Finally, the proposed architecture can be applied to many different applications, mainly those where implicit parallelism can be identified (such as in numerical computation). An outstanding feature of PRADA is its scalability that allows achieving growing performances by increasing the number of PEs.

8 Future work

The features of PRADA indicate that this architecture is a viable alternative for problems that include a large amount of processing over a small amount of data. Problems of this category are found, for instance, in numerical computing, cryptography and optimisation.

Based on the performance obtained by PRADA, there are some points for future improvement, for instance:

- improve of the connectivity between the functional blocks to increase speedup and reduce possible idle time for accessing PEs
- increase the width of internal buses and registers to allow PRADA deal with words of 64 or 128 bits
- increase the number of PEs, and thus, obtain more performance by exploring further the parallelism
- extent the number and type of instructions, incorporating more complex operations, specific for scientific computation
- adapt the system to allow dynamical reconfiguration
- implement applications in more recent and faster FPGA devices
- develop a programming environment enabling the use of a high-level language for complex applications.

Acknowledgements

The authors would like to thank Eng. Maurício Cúnico for his collaboration in this work, and the Brazilian National Research Council – CNPQ for the research grant 309262/2007-0 to H.S. Lopes.

References

Adario, A.M.S., Roehe, E.L. and Bampi, S. (1999) 'Dynamically reconfigurable architecture for image processor applications', *Proceedings of Design Automation Conference*, New York, NY, USA, pp.623–628.

Arvind, M. and Nikhil, R.S. (1990) 'Executing a program on the MIT togged-token dataflow architecture', *IEEE Transactions on Computers*, Vol. 39, No. 3, pp.300–318.

Becker, J. and Hartenstein, R. (2003) 'Configware and morphware going mainstream', *Journal of Systems Architecture*, Vol. 49, pp.127–142.

Bell, G., Gray, J. and Szalay, A. (2006) 'Petascale computational systems', *IEEE Computer*, Vol. 39, No. 1, pp.110–112.

Bittner Jr., R.A., Athanas, P.M. and Musgrove, M.D. (1996) 'COLT an experiment in wormhole run-time reconfiguration', *Proceedings of SPIE Photonics East*, Boston, MA, USA, pp.187–195.

Cavin, R., Hutchby, J.A., Zhirnov, V., Brewer, J.E. and Bourianoff, G. (2008) 'Emerging research architectures', *IEEE Computer*, Vol. 41, No. 5, pp.33–37.

Charbouillot, S., Pérez, A. and Fronte, D. (2008) 'A programmable hardware cellular automaton: example of data flow transformation', *VLSI Design*, Vol. 2008, Article id160728, pp.1–7.

Cherbaka, M.F. (1996) 'Verification and configuration of a run-time reconfigurable custom computing integrated circuit for DSP applications', MSc thesis, Virginia Polytechnic Institute and State University, USA.

Dong, Y., Dou, J. and Zhou, J. (2007) 'Optimized generation of memory structure in compiling window operations onto reconfigurable hardware', *Lecture Notes in Computer Science*, Vol. 4419, pp.110–121.

Ferlin, E.P. (2008) 'A reconfigurable parallel architecture based on dataflow implemented in FPGA', PhD thesis, Federal University of Technology Paraná, Curitiba, Paraná, Brazil.

Galanis, M.D., Dimitroulakos, G. and Goutis, C.E. (2007) 'Speedups and energy reductions from mapping DSP applications on an embedded reconfigurable system', *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 15, No. 12, pp.1362–1366.

Goldstein, S.C., Schmit, H., Budiu, M., Cadambi, S., Moe, M. and Taylor, R.R. (2000) 'Piperench: a reconfigurable architecture and compiler', *IEEE Computer*, Vol. 33, No. 4, pp.70–77.

Hardnett, C.R., Jayaraj, A., Palem, K.V. and Yalamanchili, S. (2003) 'Compiling stream kernels for polymorphous computing architectures', *Proceedings of* 12*th International Conference on Parallel Architectures and Compilation Tecniques*, New Orleans, LA, USA, pp.CDROM.

Hartenstein, R. and Kress, R. (1995) 'A datapath synthesis system for the reconfigurable datapath architecture', *Proceedings of Asia and South Pacific Design Automation Conference*, Chiba, Japan, pp.479–484.

Hartenstein, R., Kress, R. and Reining, H. (1994) 'A reconfigurable data-driven ALU for Xputers', *IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, USA, pp.CDROM.

Hennessy, J.L. and Patterson, D.A. (2006) *Computer Architecture: A Quantitative Approach*, 4*th* ed., Morgan Kaufmann, San Francisco.

Lai, X. and Massey, J.L. (1990) 'A proposal for a new block encryption standard', *Proceedings of EUROCRYPT*, pp.389–404.

Liu, Y. and Furber, S. (2005) 'A low power embedded dataflow coprocessor', *Proceedings of IEEE Computer Society Annual Symposium on New Frontiers in VLSI Design*, pp.246–247.

Lodi, A., Toma, M., Campi, F., Cappelli, A., Canegallo, R. and Guerrieri, R. (2003) 'A VLIW processor with reconfigurable instruction set for embedded applications', *IEEE Journal of Solid-State Circuits*, Vol. 38, No. 11, pp.1876–1886.

Lysaght, P. and Rosenstiel, W. (2005) *New Algorithms, Architectures and Applications of Reconfigurable Computing*, Springer, New York.

Murdocca, M.J. and Heuring, V.P. (2007) *Computer Architecture and Organization*, John Wiley & Sons, New York.

Myjak, M.J. and Delgado-Frias, J.G. (2008) 'A medium-grain reconfigurable architecture for DSP: VLSI design, benchmark mapping and performance', *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 16, No. 1, pp.14–23.

PACT XPP Technologies Inc. (2006) 'XPP-III processor overview', White paper, Version 2.0.1, July 2006, available at http://www.pactxpp.com.

Park, H., Fan, K., Mahlke, S., Oh, T., Kim, H. and Kim, H.S. (2008) 'Edge-centric modulo scheduling for coarse-grained reconfigurable architectures', *Proceedings of PACT*, Toronto, Canada, pp.633–637.

Pedroni, V.A. (2004) *Circuit Design with VHDL*, MIT Press, Cambridge.

Petersen, R.J. (1995) 'An assessment of the suitability of reconfigurable systems for digital signal processing', MSc thesis, Brighan Young University, Electrical and Computer Engineering Department, USA.

Pfister, G.F. (1998) *In Search of Clusters*, 2*nd* ed., Prentice Hall, Upper Saddle River, NJ, USA.

Pillement, S., Sentieys, O. and David, R. (2008) 'DART: a functional-level reconfigurable architecture for high energy efficiency', *EUROSIP Journal on Embedded Systems*, Vol. 2008, p.13, Article ID 562326.

Quenot, G., Coutelle, C., Serot, J., Zavidovique, B. (1993) 'Implementing image processing applications on a real-time architecture', *Proceedings of IEEE Workshop on Computer Architectures for Machine Perception*, New Orleans, MI, USA, pp.34–42.

Salomão, C., Alves, V. and Chaves-Filho, E.C. (1998) 'HiPCrypto: a high performance VLSI cryptographic chip', *Proceedings of IEEE ASIC Conference*, Rochester, NY, USA, pp.7–13.

Sankaralingam, K., Nagarajan, R., Liu, H., Kim, C., Huh, J., Burger, D., Kecler, S.W. and Moore, C.R. (2003) 'Exploiting ILP, TLP, and DLP with polymorphous TRIPS architecture', *Proceedings of* 30*th Annual International Symposium on Computer Architecture*, San Diego, CA, USA, pp.422–433.

Shibata, Y., Miyazaki, H., Ling, X. and Amano, H. (1998) 'HOSMII: a virtual hardware integrated with DRAM', *Proceedings of* 5*th Reconfigurable Architectures Workshop*, Orlando, FL, USA, pp.85–90.

Schneier, B. (1996) *Applied Cryptography*, 2nd edition, John Wiley, New York.

Šilc, J., Robič, B. and Ungerer, T. (1999) *Processor Architecture: from Dataflow to Superscalar and Beyound*, Springer-Verlag, Berlin-Heidelberg.

Singh, H.S., Lee, M.H., Lu, G., Kurdahi, F.J., Bagherzadeh, N. and Chaves Filho, E.M. (2000) 'MorphoSys: an integrated reconfigurable system for data-parallel computation-intensive applications', *IEEE Transactions on Computers*, Vol. 49, No. 5, pp.465–481.

Stallings, W. (2009) *Computer Organization and Architecture*, 8th ed., Prentice Hall, Upper Siddle River.

Swanson, S., Michelson, K., Schwerin, A. and Oscin, M. (2003) 'WaveScalar', *Proceedings of* 36*th IEEE/ACM International Symposium on Microarchitecture*, pp.291–302.

Tanenbaum, A.S. (2005) *Structured Computer Organization*, 5th edition. Prentice Hall, Upper Saddle River.

Teifel, J. and Manohar, R. (2004) 'An asynchronous dataflow FPGA architecture', *IEEE Transactions on Computers*, Vol. 53, No. 11, pp.1376–1392.

Vahey, M., Granacki, J., Lewins, L., Davidoff, D., Draper, J., Steele, C., Groves, G., Kramer, M., Lacoss, J., Prager, K., Kulp, J. and Channell, C. (2006) 'MONARCH: a first generation polymorphic computing processor', *Proceedings of* 10*th Annual Workshop on High Performance Embedded Computing*, Boston, MA, USA, pp.CDROM.

Varnagiryte, B. Zamelis, A., Olsen, O., Koch, P., Wolf, O. and Kazanavicius, E. (2002) 'A practical approach to DSP code optimization using compiler/architecture', *Ultragarsas*, Vol. 43, No. 2, pp.28–33.

Veen, A.H. (1986) 'Dataflow machine architecture', *ACM Computing Surveys*, Vol. 18, No. 4, pp.365–398.

Yiannacouras, P., Steffan, J.G. and Rose, J. (2007) 'Exploration and customization of FPGA-based soft processors', *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 26, No. 2, pp.266–277.