Journal of Circuits, Systems, and Computers

A FPGA-BASED RECONFIGURABLE PARALLEL ARCHITECTURE FOR HIGH-PERFORMANCE NUMERICAL COMPUTATION

EDSON PEDRO FERLIN[†]
HEITOR SILVÉRIO LOPES[§]
CARLOS R. ERIG LIMA[‡]
MAURÍCIO PERRETTO[◇]

[†] *Computer Engineering Department, Positivo University,*
R. Prof. Pedro V.P. de Souza, 5300, 81280-330 Curitiba (PR), Brazil
[†]*ferlin@up.edu.br,* [◇]*mperretto@up.edu.br*

[§] *Bioinformatics Laboratory, Federal University of Technology – Paraná,*
Av. 7 de setembro, 3165, 80230-901 Curitiba (PR), Brazil
[§]*hslopes@utfpr.edu.br,* [‡]*erig@utfpr.edu.br*

Received (Day Month 2008)
Revised (Day May 2010)
Accepted (Day Month Year)

Many real-world engineering problems require high computational power, especially regarding to the processing time. Current parallel processing techniques play an important role in reducing the processing time. Recently, reconfigurable computation has gained large attention thanks to its ability to combine hardware performance and software flexibility. Also, the availability of high-density FPGA (Field Programmable Gate Array) devices and corresponding development systems, allowed the popularization of reconfigurable computation, encouraging the development of very complex, compact and powerful systems for custom applications. This work presents an architecture for parallel reconfigurable computation based on the dataflow concept. This architecture allows reconfigurability of the system for many problems and, particularly, for numerical computation. Several experiments were done analyzing the scalability of the architecture, as well as comparing its performance with other approaches. Overall results are relevant and promising. The developed architecture has performance and scalability suited for engineering problems that demand intensive numerical computation.

Keywords: Computer Architecture; Parallel Processing; Reconfigurable Computing.

1. Introduction

Parallel reconfigurable architectures are computer architectures in which several processing elements work in parallel, and their logic blocks can be reconfigured (logically or functionally) to adapt the system features to a particular problem.

Reconfigurable computation [1] can override the bottleneck of Von Neumann's machines implemented with ordinary processors, allowing multiple levels of par-

allelism. It allies the performance of hardware-based solutions and the flexibility of software-based solutions. This approach can be an interesting solution for the computational resources growing required in many research areas (see, for instance, [2,3,4]).

Many complex problems are solved with sequential software-based solutions using conventional (mono-processed) hardware and cannot attain the required performance constraints, therefore justifying the research for more efficient architectures.

Reconfigurable computing systems present advantages over conventional approaches, such as: low power consumption, high processing speed, improved integration capability, flexibility and modular operation [5]. Differently from general-purpose computers, the flexibility of a reconfigurable architecture allows a better customization of the hardware to the application, allowing, for example, the exploration of the parallelism events in a computational solution, particularly in those including scientific computations. Such parallelism, massive or not, can lead to a significant reduction in the processing time, dividing the computational demands among several processing elements.

More than a decade ago, Manners and Makimoto [6] described three waves of circuits technology. In the first wave, the dominant technology was standard transistors and simple logic gates, used for customized solutions in which both algorithm and physical resources were fixed. In the next wave, microprocessors become available, and a paradigm shift from hardware to software took place. In this new paradigm, resources remained fixed, but the algorithm was variable. More recently, hardware reconfiguration started the third wave in circuits projects, establishing a migration from procedural solutions to structural solutions. This new approach is characterized by the reconfigurability of both algorithm and resources. This work is developed in the context of this third wave.

This paper presents a reconfigurable computer architecture using parallel computing concepts to obtain a scalable performance. The concept of adapting the architecture to the application is explored here. The proposed architecture can be adapted to several problems, such as numerical computation. For instance, this architecture can explore the inherent parallelism of the numerical operations required for computing differential equations such as Eq. (1), where many operations can be done simultaneously using a dataflow model.

$$\frac{d^2y}{dx^2} + 2x\frac{dy}{dx} + 2y = 0. \tag{1}$$

Differential equations, such as the one above, have operations that can be done in parallel. The the specific processing elements available in the proposed architecture can explore efficiently such possible parallelism, not achievable by sequential processing.

2. Parallel Processing

Parallel processing [7] is an efficient way to process data exploring possible simultaneous events of a software execution. The motivation for parallel processing is the possibility of increasing the computational performance for solving a complex problem using many Processing Elements (PEs). The technological speed limit imposed to sequential processing machines based on the Von Neumann architecture can be overcome by using an arrangement of PEs operating in parallel [8].

Processors with parallel architectures include a large spectrum, from single ALUs (Arithmetic and Logic Unit) to sophisticated microprocessors [9]. For instance, it is possible the use of SIMD (Single Instruction, Multiple Data) architectures, performing binary arithmetic operations (sum, subtraction) and logic operations (and, or, not) or the use of MIMD (Multiple Instruction, Multiple Data) architectures, performing complex computations, such as the Pentium® processors.

The central memory of parallel computers can be shared between processors in two ways [10]:

Physical: the same addressing space is common to all processors. In this case, if stored values in the memories are changed, all processors will use this value;

Logical: a data structure is used in such a way that, when the information is written by one of the processors, it can be read by the other.

Usually, there are two modes in which the connectivity among the several components of a computer system, such as processors and memories, can be done [11]:

Statically: the components are physically connected when the computer was built and the topology cannot be changed during use;

Dynamically: there are switching elements responsible for routing data and commands among the components. Shared memories and specific communication channels are used for routing.

The use of parallelism in computer architectures has allowed a significant increase of processing speed due to the concurrent execution of tasks. However, not only architectural features are important. The use of parallel software and how the programs can be parallelized are equally essential for achieve high performances [12].

3. Reconfigurable Computing

The idea of using reconfigurable hardware for computer systems appeared in the 1960s, but the first practical demonstration of its feasibility was only in the 1980s, when reprogrammable devices came up, such as the FPGAs (Field Programmable Gate Array) [13]. Reconfigurable devices can be considered modern solutions for computer hardware projects. They fill up the gap between ASICs (Application Specific Integrated Circuits) and conventional microprocessors [14], breaking the balance point between flexibility and performance.

Reconfigurable systems can achieve high performance with low implementation cost. Also, they override the well-known bottleneck of Von Neumann's machines implemented with ordinary microprocessors, allowing massive low-level parallelism. Reconfigurable computing associate the performance of a hardware-based solution and the flexibility of a software-based one. That is, it offers higher performance than that obtained by a software-based approach, with larger flexibility than that obtained by a hardware-based approach.

Reconfigurable computing exploits the fact that, in computational intensive tasks, most of the processing time is spent in a relatively small portion of the software. Therefore, some kind of hardware acceleration can improve significantly the performance of a processor in those applications [15].

Generally, reconfigurable computer architectures are those where reconfigurability concepts and reconfiguration techniques are intensely used [1,4]. That is, in such architectures the logical blocks, as well as the interconnection blocks, can be reconfigured to perform different functionalities. Logical blocks are understood as being the processing, storage, communication, input and output structures. Therefore, reconfigurable hardware is programmable by reconfiguration of its structure – a combination between hardware and software approaches. An algorithm structurally programmed in reconfigurable hardware is also known as configware [16]. Configware aggregate software and hardware concepts so as to explore the inherent parallelism of computational tasks.

Recently, the availability of high-density reconfigurable devices with massive interconnection capability provided a new internal organization of these devices, known as Reconfigurable Data Path [17]. This new organizational model leaded to improved parallelism and even better performance.

4. The Dataflow Model

Dataflow architectures come up by the end of the 1970s to explore the parallelism found in some program instructions [18]. Dataflow architectures use a single memory for both data and instructions, and do not use a program counter (as in conventional Von Neumann processors) [15]. Also, dataflow architectures do not manipulate variables, because values are represented by packets, denominated templates, transmitted between PEs. Each PE has the task of performing an operation using its inputs and generating an output. In this case, each operation is dependent of only two inputs. Consequently, there are no global variables nor any other external data needed, and any PE can do its job as soon as the required data is available at its inputs. The sequence of the operations is implicit to a given application and depends only of the input data.

Dataflow uses a template associated to each PE. It contains information about the operation to be done, buffers for the input data, and a list of destinations for the output. A template is similar to an instruction of a conventional microprocessor. An execution cycle consists in fetching and dispatching all ready-to-run templates,

running the templates and storing results in the appropriated destinations. Working this way, if the processing is started with a single PE, and later other PEs are added to the system, the overall performance will grow until all implicit parallelism be explored, taking advantage of the scalability provided by the approach.

As mentioned before, in the dataflow model the control flow over the operations is a function of the availability of input data for a given instruction processing. That is, the system is data-driven. A dataflow program is organized as a graph in which vertices represent instructions and edges represent the data flowing between vertices. As soon as the vertices detect that all their input edges are enabled (input data are available), they execute the programmed operation and generates output results. These results will enable further vertices. Therefore, parallelism occurs naturally in the dataflow model as the data flows throughout the graph.

An example of the inherent parallelism of the numerical computation of a differential equation is given. The dataflow graph for computing equation Eq. (1) is shown in Figure 1. We stress that this example is only to illustrate a simple computational problem with its equivalent dataflow. The corresponding software approach, written in C programming language is given in same figure just for comparison and to facilitate understanding the dataflow graph. In Figure 1, there are 16 elementary operations: 6 multiplications, 2 additions, 2 subtractions, 3 duplications ("Dup"), 1 conditional branch ("If"), 1 relational comparison ("<") and 1 stop ("Stop"). Initially, there are 5 independent vertices in the graph that can be processed simultaneously. Later, other 5 are enabled and so on, following the edges of the graph. However, things can happen without such formal synchronization since, as soon as data is available at inputs, vertices can do their operations, and these operations can take different amount of time to completion.

In the recent literature, some reconfigurable dataflow architectures can be found, for instance, [14,21,22,23]. In particular, the two architectures are worth to mention, as follows. The KressArray [24] has a matrix structure, in which operations are mapped into a cluster of PEs, named DPUs (DataPath Units). However, the control is centralized and each DPU is composed by a fixed-point ALU. Consequently, the operations can be of a single data type, thus limiting the applicability of the architecture. The COLT [25] architecture, proposed in the 1990s, has a matrix of PEs (named IFUs – Interconnected Functional Units) operating as stages of pipelines. The matrix of IFUs have to be configured using a crossbar commuter in order to implement the desired functions of the pipeline. This approach consumes a large amount of resources from the reconfigurable device. The main difference between the above mentioned architectures and the one proposed here is that ours can be applied to a larger range of problems, requesting only the re-compilation of the software for a specific dataflow graph. This is due to the fact that the architecture and, in particular, the PEs can be reconfigured to adapt to a new problem.

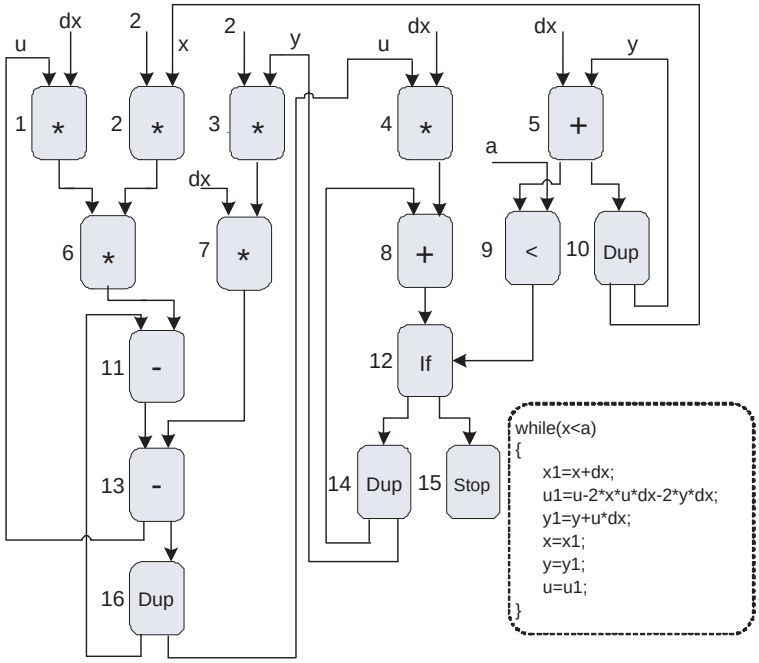

Fig. 1. Dataflow graph corresponding to Eq. (1).

5. The Parallel Reconfigurable Architecture

In this work, we propose a parallel reconfigurable architecture based on the dataflow model. In this computational model, the control is dataflow dependent, since the operations are executed as soon as input data are available. The order of the program instructions has no effect over the order of execution. This architecture can be applied to computational problems with simultaneous events, such as numerical computation.

The physical implementation of the proposed architecture was done in FPGA, allowing reconfigurability to better adaptation to a given application. Although the whole architecture can be reconfigured, it is more usual and reasonable to change the number and the structure of PEs, the width of interconnection busses, and the size of the memory. Therefore, the reconfiguration is semi-static, that is, it takes place before running, thanks to the use of a modern FPGA device.

An overview of the architecture is shown in Figure 2, composed by three elements. The Controller manages the communication between host and Dataflow Machines (DFs), as well as controls where templates are sent to, and when this takes place. Due to the nature of the tasks the Controller performs, it was implemented with an embedded processor, the Altera's NIOS II. The Switch Network is the connection element between the Controller and the DFs, and should be simple

for not consuming many resources of the device, such as logical gates. This architecture is scalable and can have the number of DF machines required by a specific application. Each DF is completely independent of the other, and parallelism arises naturally at the program level.

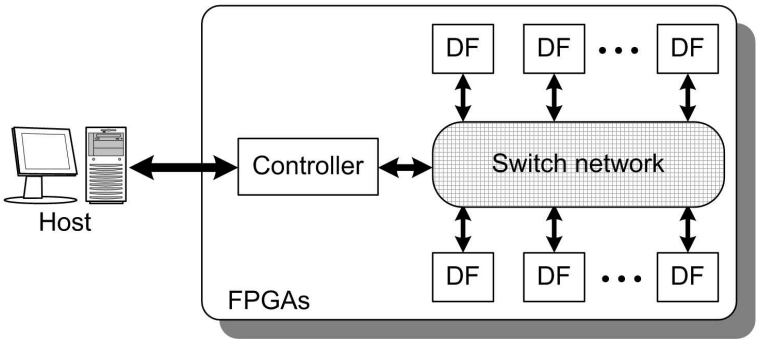

Fig. 2. Simplified diagram of the parallel reconfigurable architecture.

In this conceptual vision, a cluster of DF nodes is under the control of a Controller node. DF nodes can be different each other, having different functionalities (operations), thus allowing more flexibility to the architecture. As mentioned before, the architecture can be reconfigured as needed. This is particularly interesting for the DFs, aiming at adapting to specific application requirements. Consequently, it is possible to configure a parallel architecture either homogeneous (same DFs), as in the case of a multiprocessor, or heterogeneous (different DFs), as in the case of a multicomputer [8,20].

The nodes of the dataflow graph (see example in Figure 1) are mapped into templates. Each template corresponds to the instruction and contains all the information necessary to this execution. In our implementation, the template has the following logical structure: operation to be done – opcode (16 possible operations), two operands, containing the input data (16 bits each), two destinations to where the result of operation will be sent (8 bits each), besides other control bits (for indicating when the operand is available or which will be the destination operand, for instance). The objective of defining two destinations in each template saves one data transfer operation, since the result can be sent out to two distinct destinations in a single operation. Studies indicate that around 45% of the operations in algorithms for processing differential equations and cryptography use two destinations. Therefore, the proposed architecture can take advantage in that sort of applications [19]. The template shown in Figure 3 is mapped in a memory position of the DF machine. Therefore, in this work, each template is 57 bits long.

A DF machine is composed by a Control Unit (CU) and several Processing

Fig. 3. Basic structure of the template.

Elements (PEs), as shown in Figure 4. The CU is responsible for managing the data flow, sending templates to PEs and receiving the result of the execution of the application's dataflow graph. This graph defines the dependency among operations. In this case, the parallelism is defined at compilation time. The CU is composed by five basic elements that operate simultaneously:

Dispatch Unit (DU): Verifies if there are templates to be processed and sends them as soon as a PE is available;

Storage Unit (SU): Receives data from PEs and stores the templates;

Interface (IF): Controls the communication between the DF machine and the Controller/host by receiving templates to be processed and sending back results;

Table of Processing Elements (TPEs): Stores the current status (free or busy) of the PEs. Each position of this table corresponds to a specific PE. When the dispatch unit sends a template to a PE, its status is flagged to busy. On the other hand, when the storage unit receives the result from a PE, its status is cleared.

Template Memory (TM): Stores the operations received from the host. This memory is physically shared by the PEs by means of the dispatch and storage unities.

PEs are responsible for processing tasks. They run templates sent by the CU. Internally, PEs are composed of ALUs that effectively execute operations with data, and buffers to store templates and results. The complexity of PEs depends on the task required by the application. Usually, the internal ALUs will execute simple operations such as addition, subtraction, multiplication, comparison or conditional branch. PEs have to be as simple as possible, using minimal hardware resources (logical elements and memory) for their implementation. This allows the implementation of a large number of PEs in a reconfigurable device (FPGA), and enable the parallel execution of a large number of operations. This results in a massive parallel processing. In a PE, as soon as the input data is available, it template can be executed.

PEs are functionally autonomous, due to the fact that each PE does its own operation, independently of the other ones. This is an important feature that enables a high level of parallelism, since there is no dependency during processing, except those imposed by the application itself. However, such dependency can be minimized, as mentioned before.

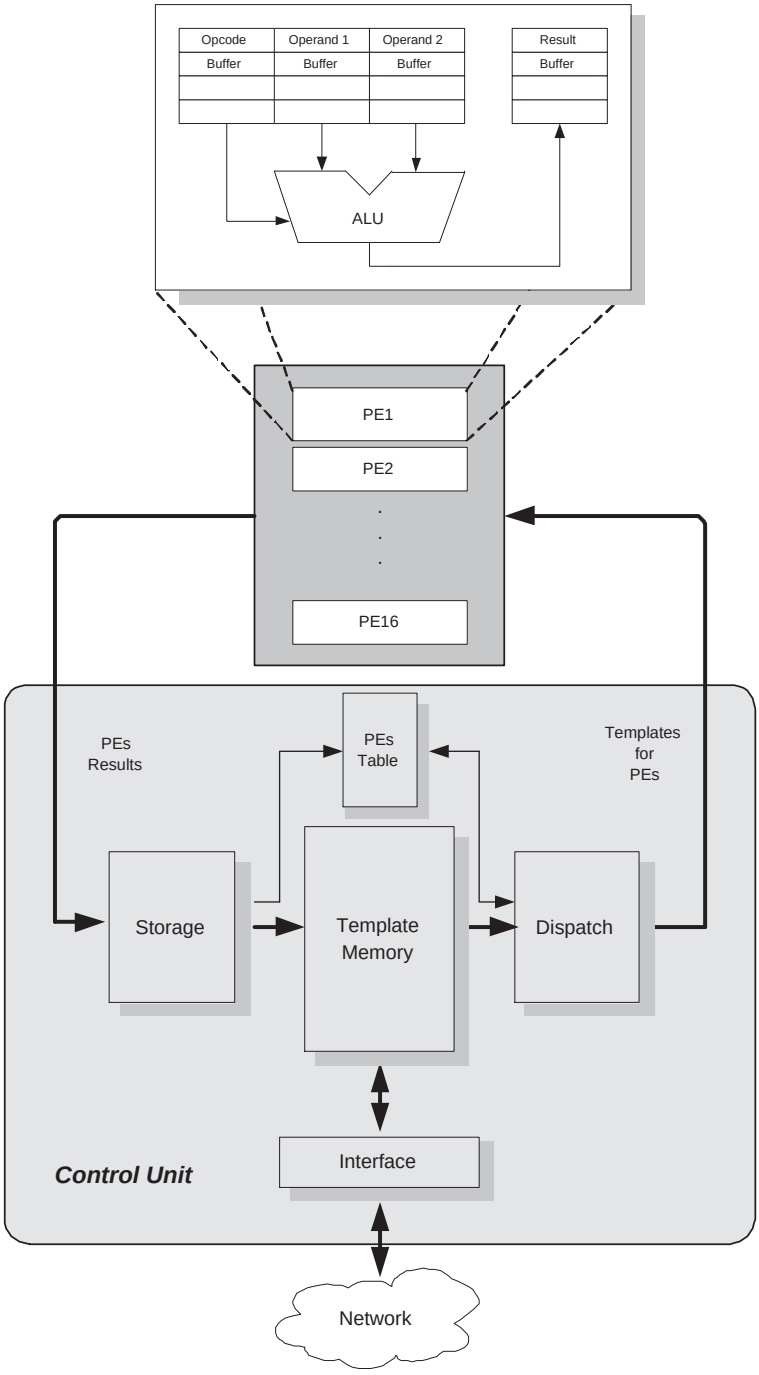

Fig. 4. A detailed view of the Dataflow (DF) machine.

The sequence of templates that composes the program, sent by the host computer, is received by the interface that stores the TM. Next, the CU starts processing the templates. The processing cycle is composed by three simultaneous processes, as follows:

- The DU sends the templates that are ready do be processed to the available PEs;
- PEs execute the operations of the templates;
- Data processed by the SU are updated in the operands of the appropriate templates.

The DU sends a template ready to be processed to a PE only when it is free (that is, the corresponding flag of the PEs' table is active), and then set its status to busy. The PEs execute their operation using the information contained in the template only (operation and operands), and then sends to the SU the result of the operation as well as the information about the destination template. When the SU receives such information, the status of the corresponding PE is cleared, enabling it for a new processing cycle. Next, the SU updates the result in the destination template, thus making it available to be processed further. This cycle is repeated until the STOP instruction is found. This makes the DU to stop fetching new templates in the TM, while waiting for the PEs and SU to finish processing their current templates. When, at last, all templates are processed, they are sent to the host by the interface.

In this architecture the components are statically connected at the implementation time and, in our particular implementation, the communication between CU and the PEs uses a parallel bus to transport both templates and results.

The proposed architecture has a superscalar structure [20], with a three-stage pipeline (Dispatch, Processing and Storage), that divides the execution of instructions in several parts. These parts are executed in parallel, each one being processed by a dedicated hardware with a specific function. Figure 5 shows the pipeline, composed by one Dispatch Unit, four PEs and two Storage Unit. A superscalar structure implicates in having functional units in the architecture each one with its own pipeline. Pipeline is a consequence of the time overlay of the execution of operations. This takes place at the different elements of the architecture when operating simultaneously in different stages. Recall that regular processors also use pipelining to overlap the execution of instructions, improving their overall performance [8]. The instruction overlap, known as ILP (Instruction-Level Parallelism), corresponds at the lowest level of parallelism in the proposed architecture.

Aiming at to verify the validity of the concepts and feasibility of the approach, we implemented an architecture with a single DF machine, having 16 PEs, as shown in Figure 4.

The architecture implemented uses a set of 16 instructions, grouped as arithmetic (addition, subtraction, multiplication, division), logical (and, or, not, xor, nor), conditional (if), relational ($=, <>, >=, <$), and miscellaneous (duplication,

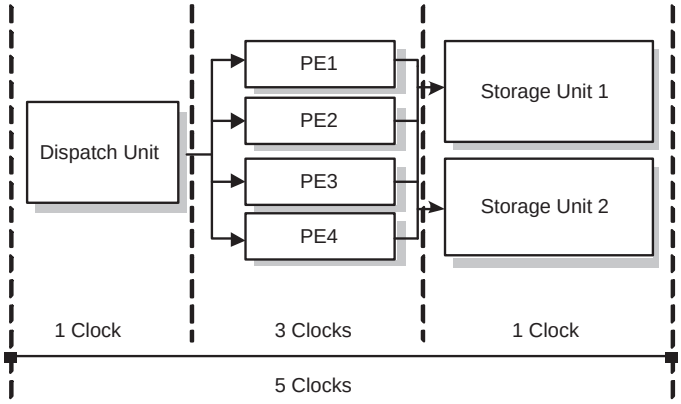

Fig. 5. Pipeline structure.

stop).

All the logic blocks of the architecture are implemented in a FPGA, including the interconnection among internal blocks. The reconfiguration of the device will occur semi-statically. In the near future, this reconfiguration will be dynamical, at execution time, especially regarding the number of processing elements. This will have direct implications in the reconfiguration time and hardware resources, since a part of the FPGA could be modified and another part could continue processing at the same time.

6. Experiments and Results

In this section, the proposed architecture is evaluated according to its performance and scalability, by using real-world applications. Also, other approaches and dedicated processors are compared with the architecture.

The proposed architecture was physically implemented in a FPGA device of the Stratix family from Altera (*http://www.altera.com*), namely, the EP1S10F780C7ES device. The implementation, with 16 PEs, required 9,697 logic elements and 10,240 memory bits, corresponding, respectively, to 91% and 1% of the resources available in the device. Each PE needs 931 logic elements – less than 8% of the device. We used the Quartus II development system from Altera and all blocks were developed in standard VHDL (Very-high-speed-integrated-circuit Hardware Description Language).

The architecture was run at 50MHz, thus having a clock cycle of 20ns. The execution time of a single template (operation) takes around 100ns. This processing time is the same for all, but division operation. Therefore, this system is capable to run 10 million of operations per second using a single PE. Consequently, a very high processing speed can be achieved with several PEs in parallel. This feature is

essential for applications that demand a high computational power, for instance, numerical computation. In such a class of applications, few different operations are done many times within the loops of a program, following the dataflow of the application.

6.1. *Evaluation of Scalability*

Depending on the number of PEs that can be implemented in a FPGA device, the processing power can be scalable to a large range. To evaluate how the processing performance of the architecture behaves as the number of PEs increase, several experiments were done. These experiments were done using an algorithm for computing a 5-tap FIR (Finite Impulse Response) filter with 16-bit fixed-point arithmetic. This algorithm, shown in Figure 6, was chosen because it has was used as benchmark for other architectures, and, thus, performance comparisons can be done. A growing number of PEs was added to the architecture until reaching 16.

```
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];
}
```

Fig. 6. 5-tap FIR algorithm.

We observed that the total processing time decreased as the number of PEs increased in the parallel structure, as shown in black-dotted curve (Real) and white-dotted curve (Ideal) of Figure 7. The ideal time is obtained by the division of the sequential execution time with a single PE by the amount of PEs. For instance, ideally with 4 PEs the time would be 1/4 of the sequential time. However, we observed a non-linear relationship in this curve, mainly due to the bottleneck created in the access to the template memory. Notwithstanding, bottlenecks are expected for any parallel architecture.

6.2. *Comparison With Other Approaches*

Using the same algorithm (5-tap FIR filter), an experiment was done with the proposed architecture (with a single PE). The algorithm was run in a general-purpose computer, a PC with Athlon XP 64 3000+ processor, running at 2.17GHz, with 512MBytes of RAM and Microsoft Windows XP operating system. In the PC, the algorithm takes 0.358μs, far below from the time taken by the proposed architecture (5.04μs). 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 should consider the number of clock cycles. For this case, the PC needed 777 clock cycles and the 1-PE-architecture needed 740 clock cycles.

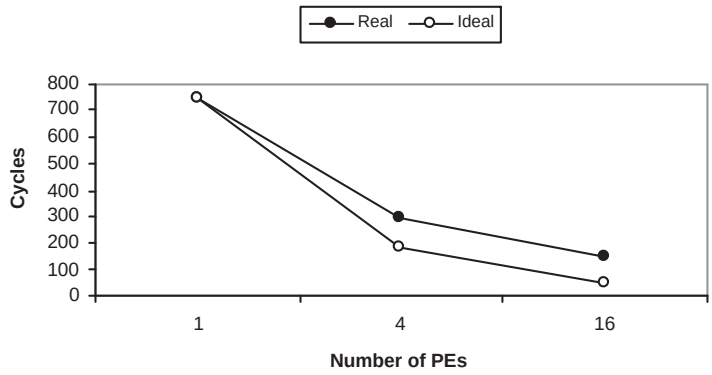

Fig. 7. Processing time (cycles) as function of the number of processing elements.

Using the same algorithm, we compared the performance of the architecture with other ones published in the literature: ROCCC (Reconfigurable Computing Compiler System) and OGMS (Optimization Generation Memory Structure for Window Operations) [27], DSP TMS 320C55X [28], a simple 8051-microcontroller, and an embedded NIOS II microprocessor. It should be stressed that the ROCC and OGMS architectures are specifically designed for digital filtering. Our architecture needed a number of clock cycles around 2.8 and 1.5 times larger than the specialized architectures and the DSP TMS 320C55X, respectively, but 2.7 and 11.6 times smaller than the NIOS II/e and the 8051, respectively. A comparison of the performance of these architectures is presented in Table 1, showing the running frequency, execution time (in clock cycles), throughput, CPI (Cycles Per Instruction), number of instructions executed, and MIPS (Millions of Instructions Per Second).

Table 1. Experiments using the FIR algorithm - execution with 1 PE.

Feature	Architecture	ROCCC	OGMS	TMS 320C55X	8051	NIOS II/e
Frequency (MHz)	50	94	238.664	200	12	50
Execution Time (Clocks)	740	262	263	504	8559	1986
Throughput	0.34	0.96	0.96	0.50	0.03	0.13
CPI	5	*	*	*	*	1.36
Number of Instructions	148	*	*	*	713	2701
MIPS	10	*	*	400	1	45

* information not available.

Another experiment was done comparing the execution time of the architecture with 4 and 16 PEs. This comparison, shown in Table 2, was done with another reconfigurable architectures, KressArray [24] and COLT [29], previously mentioned. In this table, it is observable that the executions using of our architecture with 4PEs

and 16PEs needed less time (in clock cycles) than KressArray and COLT. This fact shows the potential of the proposed architecture, as the number of PEs is increased.

Table 2. Experiments using the FIR algorithm - execution with n PEs.

Feature	Architecture 4PEs	Architecture 16PEs	COLT	KressArray
Number of PEs	4	16	16	24
Frequency (MHz)	50	50	50	25
Execution Time (Clocks)	296	148	496	558
MIPS	20	41	*	*

* information not available.

6.3. *Comparison With Dedicated Processors*

A third experiment was done considering the execution of a real FIR filter, from the BDTi (Berkeley Design Technology Inc) package [30]. This package corresponds to a set of programs for digital signal processing. The BDTi Real Block FIR Filter benchmark is used for voice processing and consists of a FIR filter with 15 taps, processing 40 input samples. For this experiment we used 8 PEs/Units in order to have a fair comparison with the other approaches (ADI ADSP-TS201S, Motorola MSC8103, TI TMS320C6414) that use the same number of PEs.

Table 6.3 shows the results of the experiment with the BDTi Real Block FIR Filter. We compared the number of processing units, running frequency, power consumption (Power), and execution time (Clocks). In this table it is observed that the power consumption of the proposed architecture is as low as most of the other ones, but, on the other hand, it needs less processing cycles (129) to do the job. It should be noted that the other architectures to which we compared are specific for digital signal processing. Figure 8 emphasizes the differences between approaches regarding the number of clock cycles.

Table 3. Comparison of architectures using the BDTi Real Block FIR Filter benchmark.

Feature	Architecture	TMS320C6414	ADSP-TS201S	MSC8103
Number of PEs/Units	8	8	8	8
Frequency (MHz)	50	720	600	300
Execution Time (Clocks)	129	202	160	183
Power (W)	0.65	0.65	2.18	0.65

7. Conclusions and Future Work

Three groups of experiments were done to evaluate the performance of the proposed reconfigurable architecture, as well as to compare it with other approaches.

Results of the experiments showed that using only 4 PEs the processing time was 60% above the estimated ideal time, while using 16 PEs, this value was above

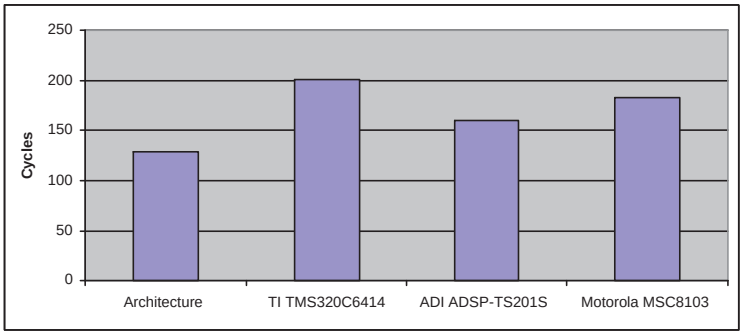

Fig. 8. Processing Time (cycles) for the BDTi Real Block FIR Filter Benchmark.

180%. However, it should be stressed that this is a real-world application and, consequently, it presents many dependencies among operations, precluding the proposed architecture to reach its idealized peak performance. This result is justified due to the bottleneck imposed by the access to the template memory.

Another important performance measure for analyzing parallel machines is the speedup [20]. Speedup is the ratio between the execution time with a single PE, and the execution time for several PEs in parallel. In the experiment with 16 PEs, a speedup of 5 was obtained, corresponding to 31% of the ideal value. This is due to the parallelization constraints implicit in the application code and that can be explicable by the Amdhal's Law [26].

The comparison of performance running the 5-tap FIR algorithm in our architecture (with a single PE) and in other architectures showed that it is faster than conventional microprocessors (including the DSP). As expected, the proposed architecture was slower than the architectures specifically designed for digital filtering. However, it should be taken into account that those results would be quite different in favor of the proposed architecture if using 16 PEs. Also, thanks to the dataflow approach, the number of instructions necessary to execute the algorithm was significantly smaller when compared with the other conventional architectures.

An important comparison was done with other parallel reconfigurable architectures (COLT and KressArray). The results showed that our approach is around 1.8 to 3.8 times faster, depending on the number of PEs used. Even with only 4 PEs, our architecture achieved better performance than the other architectures using much more parallel PEs.

The experiment with a real FIR filter benchmark showed that the proposed architecture needs less processing cycles than other specific digital signal processing architectures (TMS320C64X, ADSP-TS201S e MSC8103), while consuming the same power. This fact suggests that our proposed architecture could be used efficiently for such digital signal processing task. On the other hand, our architecture was run at 50 MHz, which is significantly smaller than the other architectures. Since

this speed is due to the technological limitation of the device used in the current implementation, it is expected that future versions could run at higher frequencies, thus leading to even higher performance.

Besides the parallelism of execution at the PEs level, and the low-level Instruction-Level Parallelism (ILP) – see section 5, the use of a pipeline structure allowed an additional level of parallelism in the execution, taking advantage of the simultaneous operation the internal units (Dispatch, PEs and Storage). The joint effect of all these levels of parallelism contributed to efficiency of the architecture.

Although the use of buffers in PEs and parallel busses helped to minimize the bottleneck in accessing the internal units, there are still another bottlenecks in the architecture, as inferred from the plot of Figure 7. This issue will be focused in future works.

Current results can be considered relevant. We emphasize the fact that the number of PEs can be increased up to the limit established by the FPGA resources. Besides, it is possible to increase the complexity of each PE by including operations that require a larger number of clock cycles to be done. It is obvious that a tradeoff between the complexity of PEs and their number must be established. We believe that, as more powerful FPGA devices become available, the more the proposed architecture becomes feasible and interesting for complex numerical problems.

Overall, the results of the experiments suggest the appropriateness of the architecture for solving intensive numerical problems such as in digital signal processing and other problems found scientific computation.

There are some issues that shall be addressed in future developments so as to improve the architecture, for instance: improvement of the dataflow machines to increase performance, and adaptation of the PEs to operate with floating point arithmetic.

References

1. S. Hauck, A. DeHon, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computing* (Morgan Kaufmann, San Francisco, 2008).
2. H.S. Lopes, C.R.E. Lima, N.J. Murata, A configware approach for high-speed parallel analysis of genomic data, *J. Circuit Syst. Comp.* **16** (2007) 1–15.
3. S. Leibson and J. Kim, Configurable processors: a new era in chip design, *IEEE Comput. Magazine*, July (2005) 51–59.
4. P. Lysaght and W. Rosenstiel (eds), *New Algorithms, Architec. and Applications for Reconf. Computing* (Springer, New York, 2005).
5. S.A. Ito and L. Carro, A comparison of microcontrollers targeted to FPGA-based embedded applications, *Proc. 15th Symp. Integrated Circuits and Syst. Design* (2000), pp. 397–402
6. D. Manners and T. Makimoto, *Living with the Chip* (Chapman & Hall, New York, 1995)
7. J. Dongarra *et al*, *Sourcebook of Parallel Computing* (Morgan Kaufmann, San Francisco, 2003).
8. J.L. Hennessy and D.A. Patterson, *Computer Architecture: a Quantitative Approach.* 4rd edition (Morgan Kaufmann, San Francisco, 2007).

9. A.A. Jerraya and W. Wolf, *Multiprocessador Systems-on-Chips* (Morgan Kaufmann, San Francisco, 2005).
10. W. Stallings, *Computer Organization and Architecture*, 5^{th} edition (Prentice Hall, New Jersey, 2000).
11. M.J. Murdocca and V.P. Heuring, *Principles of Computer Architecture* (Prentice Hall, New Jersey, 2000).
12. E.P. Ferlin, Evaluation of Automatic Parallelization Methods, M.Sc. thesis, Physics Institute, University of São Paulo at São Carlos (1997).
13. A.S. Berger, *Hardware and Computer Organization – The Software Perspective* (Elsevier, Amsterdam, 2005).
14. Hartenstein, R. A decade of reconfigurable computing: a visionary retrospective. In *Proc. IEEE Conf. Design, Autom. and Test in Europe*, 2001, pp. 642–649.
15. J. Silc, B. Robic and T. Ungerer, *Processor Architecture: From Dataflow to Superscalar and Beyond* (Springer-Verlag, Berlin, 1999), pp. 307–323.
16. J. Becker and R. Hartenstein, Configware and morphware going mainstream, *J. Syst. Architec.* **49** (2003) 127–142.
17. R. Vemuri and E. Randolph, Configurable computing: technology and applications, *IEEE Comput. Magazine.* **33** (2000) 39–40.
18. A.H. Veen, Dataflow Machine Architecture, *ACM Comp. Surv.* **14** (1986) 365–396.
19. E.P. Ferlin, A reconfigurable parallel architecture based on dataflow implemented in FPGA, Ph.D. thesis, Federal University of Technology - Paraná (2008).
20. A.S. Tanenbaum, *Structured Computer Organization*, 5^{th} edition (Pearson Education, New Jersey, 2006).
21. V. Bhaskar, A hybrid closed queuing network model for multi-threaded dataflow architecture. *Computers and Electrical Engineering.* **31** (2005) 556–571.
22. J.M.P. Cardoso, Self loop pipelining and reconfigurable dataflow arrays, *Proc. 3^{rd} Int. Workshop on Computer Syst., Architec., Modeling and Simul. LNCS.* **3133** (2004) 234–243.
23. S. Swanson *et al*, The microarchitecture of a pipelined WaveScalar processor: an RTL-based study, *CSE Technical Report TR-2004-11-02*, University of Washington (2005).
24. R. Hartenstein, R. Kress, and H. Reining, A reconfigarable data-driven ALU for Xputers, *IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, USA, april 1994, CDROM.
25. R.A. Bittner Jr, P.M. Athanas and M. Musgrove, Colt: an experiment in wormhole run-time reconfiguration, *Proc. SPIE Photonics East*, Boston, MA, USA, November 1996, pp. 187–195.
26. T.L. Casavant, P. Turdik and F. Plasik (eds), *Parallel Computer Theory and Practice* (IEEE Press, Los Alamitos, 1996).
27. Y. Dong *et al*, Optimized generation of memory structure in compiling window operations onto reconfigurable hardware, *Proc. 4^{th} Int. Workshop on Applied Reconf. Computing. LNCS.* **4419** (2007) 110–121.
28. B. Varnagiryte, A. Zamelis, O. Olsen, P. Koch, O. Wolf and E. Kazanavicius, A practical approach to DSP code optimization using compiler architecture, *Ultragarsas*, vol. 43, n. 2 (2002), pp. 28–33.
29. M.F. Cherbaka, Verification and Configuration of Run-Time Reconfigurable Custom Computing Integrated Circuit for DSP Applications, *M.Sc. thesis*, Virginia Polytechinic Institute and State University (1996).
30. Berkeley Design Technology, A BDTi Analysis of the Texas Instruments TMS 320C64X. *BDTi*, www.BDTi.com (2004).