# A Partition Model using Partial Reconfigurable Hardware for ChipCflow Project

Francisco Souza Junior[1] Jorge Luiz e Silva[2]

*Resumen*— **In this paper, the partition model, that was designed in a field programmable gate array (FPGA) with partial reconfiguration, as a prototype of a static dataflow architecture is discussed. The ChipCflow Project using a dataflow graph with partition of the graph in a partial reconfigurable system shows the potential for high computation rates. The model of partition was validated and results are presented to the end of this paper.**

*Palabras clave*—**Dataflow graph partition, Algorithm Acceleration, Partial Reconfiguration, dataflow architecture.**

## I. Introduction

With the advent of reconfigurable computing, basically using a Field Programmable Gate Array(FPGA), researchers are trying to explore the maximum capacities of these devices, which are: flexibility, parallelism, optimization for power, security, real time applications and partial reconfiguration [5], [15].

Because of the complexity of the applications and the large possibilities to develop systems using FPGAs, many applications to converts algorithm into these devices associated with a General Purpose Processor (GPP) using high level language like C and Java is one of the challenges for researchers nowadays, especially for accelerating algorithms [11], [12].

Another problem is the size of the system where the hardware should be optimized to execute the applications.

The main aim of this project was to accelerate the algorithms witch convert parts of programs written in C language into a dataflow model using a partition model implemented in a FPGA with partial reconfiguration.

The paper is organized as follows. Related works is described in the section II. The Dataflow Graph Model is discussed in the section III. In the section IV are presented the Model of Partition in a Partial Reconfiguration. Section V shows the results of implementations. Section VI conclude the paper and presents future works.

## II. Related Works

The dataflow graph model and its architecture was first researched in the 1970s and was discontinued in the 1990s [1], [4], [9], [10]. Nowadays, it is a topic of research once more, mainly because of the advance of technology, particulary with the advent of the FPGA [2], [9], [15].

[1]Department of Computer Systems, University of Sao Paulo, e-mail: fsjunior@icmc.usp

[2]Department of Computer Systems, University of Sao Paulo, e-mail: jsilva@icmc.usp.br

Because the dataflow model has an implicit parallelism and the FPGA is composed by parallel circuits, the dataflow model applied to a FPGA has the perfect combination to execute applications which also have parallelism in their execution [9]. However, as applications become more complex, software development is only possible using high level language such as C or Java [3] although only parts of the program will be executed directly into the hardware. Thus, several tool have been developed to convert C into a hardware using VHDL language [6], [7], [8].

In order to analyze the data dependence, many of these systems generate an intermediate dataflow graph for pipeline instructions. The optimizations, using several techniques such as loop unrolling, are concluded and finally a reconfigurable hardware using the VHDL language is generated. The hardware generated using these tools consists of coarse grain elements or assembler instructions for a customized processor as Picoblase or Nios from Xilinx and Altera respectively [16].

Another important resource nowadays is Partial reconfiguration that have been studied for researches since 1990s basically proposing a solution for limitation area into the FPGA [5], [18].

In our approach, a fine grain instruction using VHDL to implement a dynamic dataflow architecture, consisting of various nodes of processing elements and arcs to connect those nodes in a graph, with tags for the data, using partition for the graphics as a solution for the hardware limitation is used to accelerate algorithms.

## III. The Dataflow Graph Model

In the Asynchronous Dataflow Graph project developed by Teifel [15], the asynchronous system is a collection of concurrent hardware processes that communicate with each other through message-passing channels. These messages consist of atomic data items called tokens. Each process can send and receive tokens to and from its environment through communication ports. In the Teifel project, asynchronous pipelines are constructed by connecting these ports to each other using channels, where each channel is allowed only one sender and one receiver. Since there is no clock in an asynchronous design, processes use handshake protocols to send and receive tokens via channels.

In Figure 1 Teifel describes an equation converted into a dataflow graph in three different situations: (a) a pure dataflow graph, (b) a token-based asynchronous dataflow pipeline and (c) a clocked dataflow pipeline.

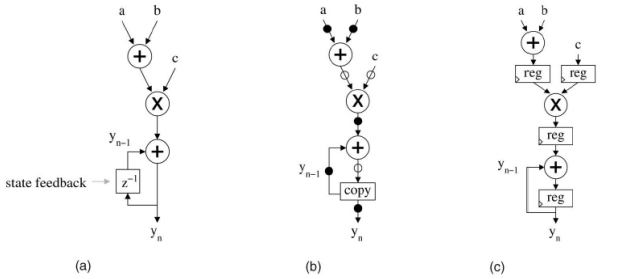


Fig. 1. Computation of y$n$=y$n$-1+c(a+b):(a) pure dataflow graph, (b) token-based asynchronous dataflow pipeline (filled circles indicate tokens, empty circles indicate an absence of tokens), and (c) clocked dataflow pipeline.

In our project, a collection of concurrent hardware processes that communicate with each other, but using a parallel bus with bits for data and bits to control the communication process in a synchronous system of communication as described in part (c) of the Figure 1, is also used.

### *A. Dataflow Computations*

In the dataflow graph for ChipCflow Project, a traditional dataflow model described in the literature, where a node is a processing element and an arc is the connection between two elements, is used [1], [2], [4], [9], [10]. A data bus and a control bus to execute the communication between the operators were implemented. The static dataflow graph model, where only one item of data can be in an arch was developed.

In Figure 2, a basic operator and its data buses and control buses for communication are described. The signal data *a*, *b* and *z* in Figure 2 are 16-bit data traveling through the parallel buses. The signals *stra*, *strb*, *strz*, *acka*, *ackb* and *ackz* are 1-bit control data to control communication between operators.

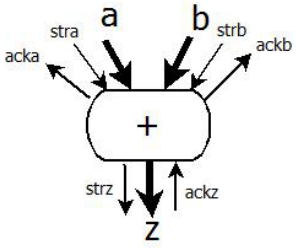


Fig. 2. The basic operator with its data buses and control buses.

The communication process between operators is described in Figure 3. As can be clearly seen in the figure, a sender operator and a receiver operator have two input data buses *a* and *b*, one output data bus *z* and its respective control signals *stra*, *strb*, *strz*, *acka*, *ackb* and *ackz*. Each of input data bus and output data bus is connected to a register to store a receiving item of data and to store a sending item of data, represented by rectangles with rounded edges *a, b* and *z* in the figure. The output data bus *z* from the sender operator is connected to input data bus *a* from the receiver operator, the output control signal *strz* from the sender operator is connected to the input control signal *stra* from the receiver operator and the input control signal *ackz* from the sender operator is connected to the output control signal *acka* from the receiver operator.

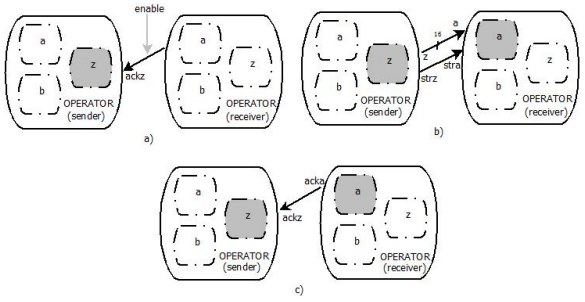


Fig. 3. The communication process: a) enabling the communication, b) sending an item of data, c) Acknowledging an item of data.

A "logic-0" in the signal *ackz* informs to the sender operator that the receiver operator is ready to receive data. A "logic-1' in the signal *ackz* informs to the sender operator that the receiver operator is busy. A "logic-1" in the signal *stra* informs to the receiver operator that an item of data is ready to be sent to it from the sender operator. A "logic-0' in the signal *stra* informs to the receiver operator that the sender does not have an item of data to be sent to it.

To initiate the communication process, an *enable* signal with a "logic-0" to the *ackz* connected to the sender, is set, Figure 3a. When the receiver operator is ready to receive data, a "logic-1" in the *stra* strobes an item of data to the input data bus *a* in the receiver operator, Figure 3b. Consequently, a "logic-0" in the *acka* acknowledges that the item of data *a* was received, Figure 3c.

### *B. The Dataflow Operators*

The dataflow operators were the traditional operators described by Veen in [10], which are: copy, non deterministic merge, deterministic merge, branch, conditional and primitive operators (add, sub, mul, div, and, or, not, etc.).

In order to execute the computation of an operator it is necessary that an item of data is presented in all its input buses of data. In Figure 4 operators are described where filled circles indicate items of data and empty circles show an absence of items of data and the situation of the operator before computation and after computation [15].

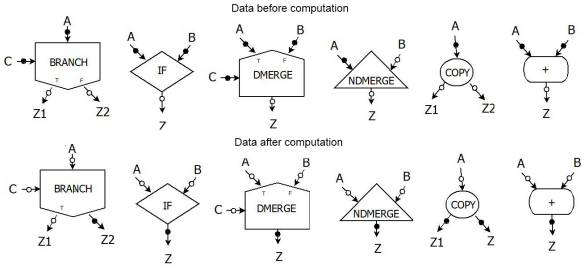


Fig. 4. The operators.

The functional execution of dataflow operators is described below:

1. *Copy*: This dataflow node duplicates an item of data to two receiver operators. It receives an item of data in its input data bus and copies the item of data to two output data buses.
2. *Primitive*: This dataflow node receives two item of data in its input data buses, computes the primitive operation with these two items of data and generates the result sending it to the output data bus. Operators such as add, sub, multiply, divide, and, or, not, if, etc., are implemented in the same way.
3. *Dmerge*: This dataflow node performs a two-way controlled data merge and allows an item of data to be conditionally read in input data buses. It receives a TRUE/FALSE item of data to decide what input data $a$ or $b$ respectively to send to the output data $z$
4. *NDmerg*: This dataflow node performs a two-way not controlled data merge and allows an item of data to be read on input data buses. The first data to arrive into the Ndmerge operator from input $a$ or $b$ is sent to the output data $z$.
5. *Branch*: This dataflow node performs a two-way controlled data branch and allows the item of data to be conditionally sent on to two different output buses. It receives a control TRUE/-FALSE item of data to decide what output data $t$ or $f$ respectively to transfer the input data $a$.

## IV. The Model of Partition in a Partial Reconfiguration

For complex application where the hardware could be prohibitive, depends on the size of the hardware, a partial reconfiguration could be a solution [5], [18].

In the ChipCflow Project, that is based on a dataflow architecture, several operators are connected by arcs to implement the application. Depends on the applications, the complete dataflow graph could not be supported by the bigger FPGA existing. In this case, a partial reconfiguration to implement the system could be used [13] [14].

To implement the partial reconfiguration for the dataflow graph, instances as a subgraph of the original dataflow graph was defined. The model of partition for ChipCflow Project consists of those instance for implementation.

In the Figure 5, an example of the partition is described, where two instances of the dataflow subgraph are defined as a partition. This partition can be partial reconfigured in the different area of the FPGA and in different moments of the execution.

As can be clearly seen in the Figure 5, a *new tag area* is an operator that generate new tag for each data coming into the instances. The first data generated for the *new tag area* operator occupy the instance 1 located in the left side of the Figure 5, that was defined as a partition. The instance 2 located in the right side of the Figure 5 is activated when a second data is generated for the *new tag area* operator and then a partial reconfiguration to receive the data for instance 2 is generated, and so on.

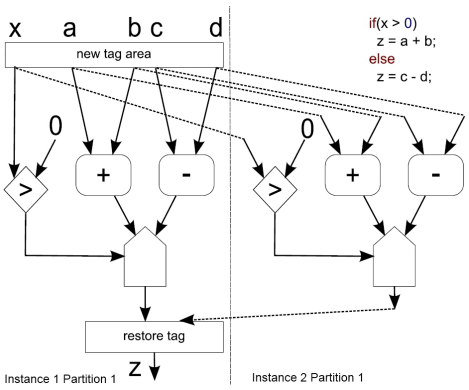


Fig. 5. Two Instances of the same Partition.

### A. Partial Reconfiguration into a FPGA

There are few FPGAs that support partial reconfiguration. The main company to produce this kind of device is Xilinx with the first one FPGA XC6200 followed for Virtex II, Virtex II pro, Virtex 4, Virtex 5, Virtex 6 and Virtex 7 all from Xilinx [17]. The Xilinx Virtex FPGA was the platform considered in this paper.

In Figure 6 an basic partial reconfiguration architecture into a FPGA is described. As can be clearly seen in figure, the FPGA is divided in PRR (Partially Reconfigurable Region) with limited reconfiguration area, associated with a static region and *bus macros* that are used to communicate the static region with the PRRs [17].

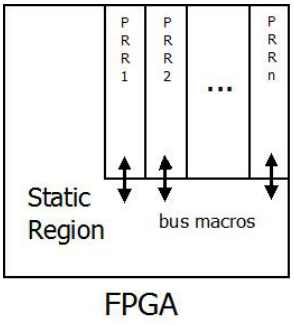


Fig. 6. Basic Partial Reconfigurable Architecture for a FPGA.

To reconfigure a specific PRR, a bitstream with respective address for that PRR is used. The other parts of the FPGA still running.

As the PRR has a limited reconfiguration area, the partition of the dataflow graph, with the maximum quantity of operators, was defined that should be partitioned into a PRR. The partition architecture, its limits and the implementation results are describe in the next sections.

### B. The Architecture for Partition

The basic organization for the partition proposed in this paper is described in the Figure 7. As can be clearly seen in the figure, there are various PRRs, a data bus distributed by the PRRs, a control bus

with a schedule that is used to control the access of PRRs to the data bus, finally a input output block to control input and output data from the FPGA.

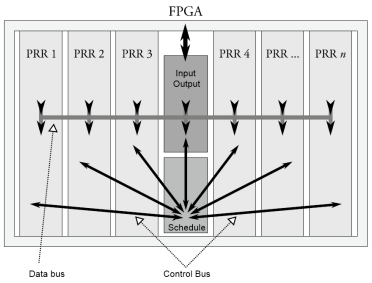


Fig. 7. The basic Organization of Partition.

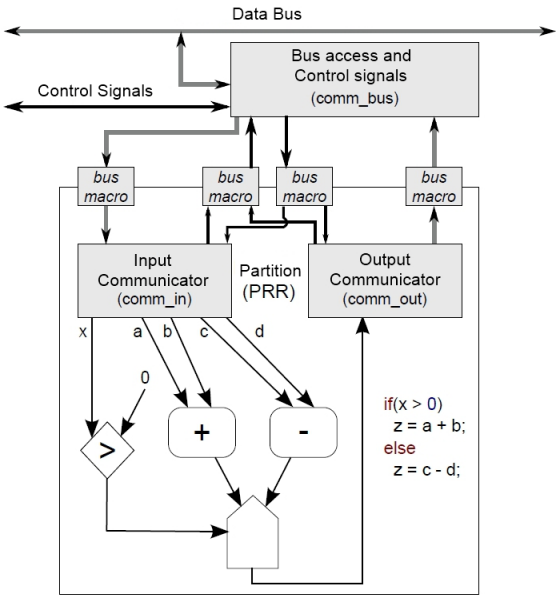


Fig. 8. A Detailed Architecture of the Partition.

In Figure 8 a detailed architecture of the partition is described where the same example of partition described in Figure 5 is used. As can be clearly seen in the Figure 8, there is a *Bus access and Control signals* block that control the access of partition to the data bus. This block is defined in static region of the FPGA. The instances, defined into the PRRs, are connected with this block through *bus macros* using the *Input communicator block* and the *Output communicator block*. When one partition send a request for schedule to send data to the data bus, the schedule commands the steps to execute that requisition.

In Figure 9 a protocol used by schedule to control the communications between partitions is described. As can be clearly seen in the figure, the field *Synchronous* is used to informs a partition that there is a data coming into it. The field *Activation*, *Iteration* and *Nesting* are parts of the data traveling through the dataflow graph. The field *Partition* and *Arc* are used for schedule to control the communication.

| Synchronous | Partition | Activation | Iteration | Nesting | Arc | data |
|---|---|---|---|---|---|---|
| 8 bits | 8 bits | 8 bits | 8 bits | 4 bits | 4 bits | 32 bits |

Fig. 9. The Protocol Frame.

## V. Experimental Results

The Fibonacci Algorithm described in the Algorithm 1 was implemented using a (xv2p30-7ff896) Virtex II FPGA from Xilinx and synthesized in ISE 9.1. The dataflow graph for Fibonacci algorithm is described in the Figure 10.

**Algorithm 1** Calculate Fibonacci

$a \Leftarrow 0$
$b \Leftarrow 1$
$t \Leftarrow 0$
**for** $i = 0$ to $n - 1$ **do**
  $t \Leftarrow a + b$
  $a \Leftarrow b$
  $b \Leftarrow t$
**end for**

As can be clearly seen in Figure 10 there are two different areas in the figure. The first one is located in the left side on figure, before the *new tag area* box. This area is used just to initialize the variables for the Fibonacci sequence. The second area is the other part of the figure but after the *new tag area* box and it is used just to control the index $i$ and the generator for the Fibonacci sequence.

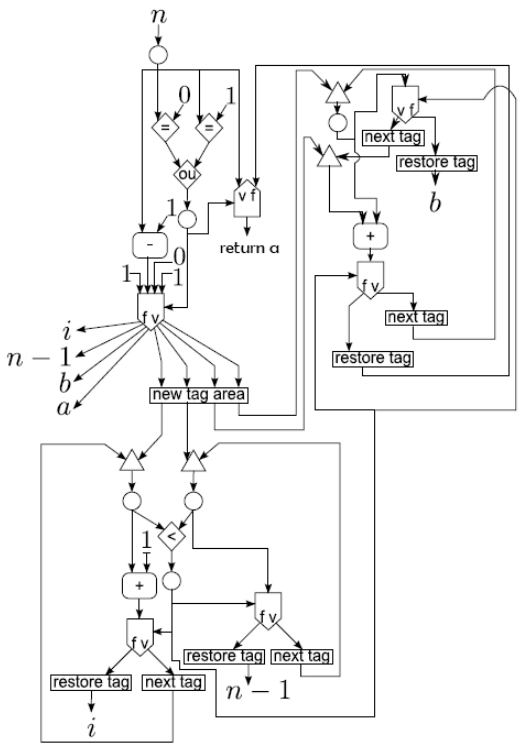


Fig. 10. Dataflow Graph for Fibonacci Algorithm.

The main aspect to define the partition for the Fibonacci dataflow graph was the limitation area for PRRs. In the Figure 11 the first partition, to be configured in the static region into the FPGA is de-

scribed. In this project, to validate the partition model, just two partitions were defined.

As can be clearly seen in Figure 11 the signals *n* and *a* are inputs of the partition *p1* and the signals *i(p2), n(p2), b(p2)* and *a(p2)* are outputs of the partition *p1* and input of the partition *p2*. A *bus macro* is used for the signal *i(p2), n(p2) b(p2)* and *a(p2)* to communicate with the partition *p2*.

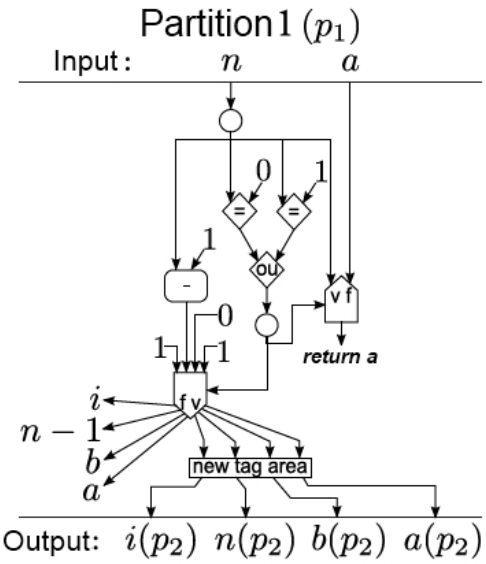


Fig. 11. The first partition for Fibonacci Algorithm.

In the Figure 12 the second partition is described. In this case, supposing to implement Fibonacci with *n=3*, two activations for partition *p2* were defined. As can be clearly seen in Figure 12 the signals *i, n, b* and *a* are inputs of the partition *p2*. The signals *i(p2), n(p2), b(p2), a(p2), a(p1)* and *b(p2)* are outputs of the partition *p2* and input of the second activation of the partition *p2* and just one signal *a(p1)* is an input for the partition *p1*. For all the signals, to communicate with the partition *p1*, the *bus macro* was used and another activation for partition *p2* was generated.

A logic representation of the partitions into the FPGA is described in Figure 13. As can be clearly seen in the figure, the part represented for the green color is the partition *p1*, the part represented for the blue color is the first activation for partition *p2* and the part represented for the red color is the second activation for the partition *p2*.

To compare the implementation of the partition model with a general purpose processor, the Fibonacci algorithm was executed in C using a Dell Vostro 1310 computer with Intel Core 2 Duo T5670 - 1.8Ghz inside. The MingW32 compiler was used [19] and the results was compared with the model of partition implemented.

Two methods were used to define the comparisons. The method 1 consists of use the Windows API with *QueryPerformanceCounter* [20] and the method 2 consists of use Assembly rdtsc from Intel Corporations [21]. In Figure 14 the results of comparison is

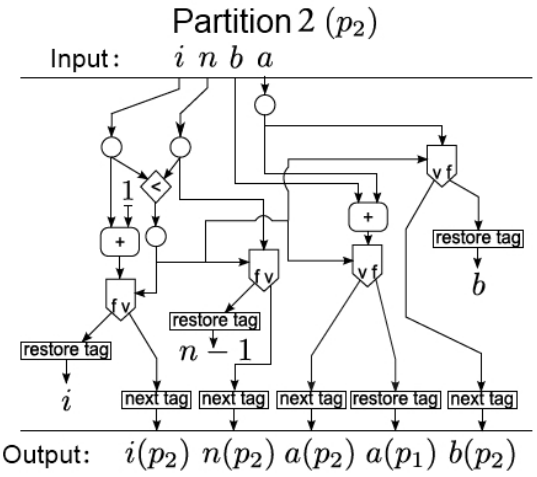


Fig. 12. The second partition for Fibonacci Algorithm.

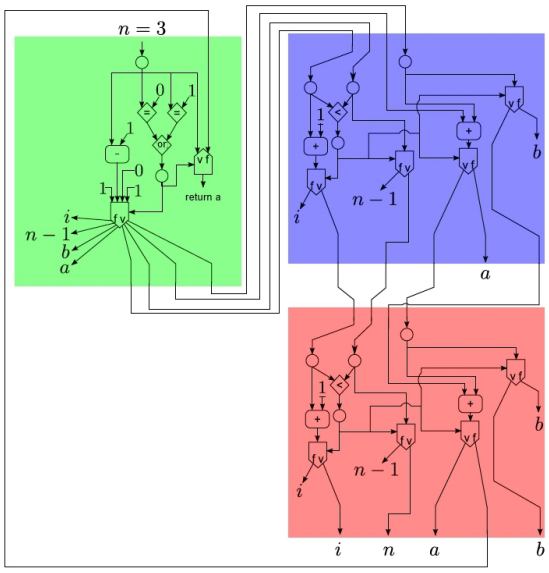


Fig. 13. The Color Representation for the Partitions.

described.

As can be clearly seen in Figure 14 the implementation generated based on Partition has less speed than the two implementation using C code. Comments is described as followed.

## VI. Conclusion

The Model of Partition has less speed than the C code using method 1 and method 2. However, the main aim in this project was to validate the implementation model likely to convert algorithms into the dataflow graph and into a VHDL using different partitions. Taking this into account, the partition model become one more solution for hardware capacity in FPGA. The benchmark used in this paper basically perform operation using vector, but it is very important to explore the maximum parallelism of the dataflow graph using real parallel applications. Future work would be to develop a module to convert C directly into a VHDL, associated with the FPGA and to implement a dynamic dataflow model using

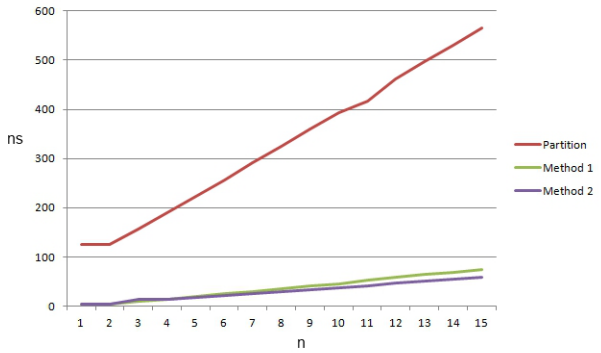


Fig. 14. The Speedup of C code versus Partition implementation.

the partition model to obtain a better performance than the results described in this paper.

## Referencias


[1] Arvind (2005) Dataflow: Passing the token, *The 32th Annual International Symposium on Computer Architecture (ISCA Keynote)*, ACM, Madison, USA, pp. 1-42.

[2] Cappelli, Andrea and Lodi, Andrea and Mucci, Claudio and Toma, Mario and Campi, Fabio. (2004) A Dataflow Control Unit for C-to-Configurable Pipelines Compilation Flow, *IEEE Sumposium on Field-Programmable Custom Computing Machines FCCM'04*, IEEE, USA, pp. 323–333.

[3] Cardoso, J., H. Neto (2003) Compilation for FPGA Based Reconfigurable Hardware, *IEEE Design Test of Computers*, IEEE, pp. 65–75.

[4] Dennis, Jack B. and Misunas, David P. (1974) A preliminary architecture for a basic dataflow processor, *Computer Architecture News - SIGARCH'74*, ACM, USA, pp. 126–132.

[5] Hauck, S. (2000) The Roles of FPGAs in Reprogrammable Systems, *Proceedings of the IEEE*, IEEE, pp. 615-638.

[6] ImpulseC (2005) *Impulse Accelerated Technologies, Inc-ImpulseC From C software to FPGA hardware* , ImpulseC.

[7] Spark (2004) *User Manual for the SPARK Parallelizing High-Level Synthesis Framework Version 1.1*, Center for Embedded Computer Systems.

[8] Suif (2006) *The Stanford SUIF Compiler Group* , Suif compiler system.

[9] Swanson, S. and Schwerin, A. and Mercaldi, M. and Petersen, A. and Putnam, A. and Michelson, K. and Oskin, M. and Eggers, S. J. (2007) The Wavescalar Architecture, *ACM Transactions on Computer Systems*, ACM, pp. 4:1-4:54.

[10] Veen, A. H. (1986) Dataflow Machine Architecture, *ACM Computing Surveys*, ACM, pp. 365–396.

[11] Chen, Z. and Pittman, R. N. and Forin, A. (2010) Combining multicore and reconfigurable instruction set extensions, *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays - FPGA'10*, ACM, USA, pp. 33–36.

[12] Hefenbrock, D. and Oberg, J. and Thanh, N. T. N. and Kastner, R. and Baden, S. B (2010) Accelerating Viola-Jones Face Detection to FPGA-Level Using GPUst, *18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, ACM, USA, pp. 11–18.

[13] Junior, F. S and e Silva, J.L. and Sanches, L. and Astolfi, V. (2010) Research and Partial analysis of overhead of a partition model for a Partially Reconfigurable hardware in a data-driven machine - chicflow *Programmable Logic Conference (SPL)*, IEEE, USA, pp.191–194.

[14] Silva, J.L. and Lopes, J. (2010) A dynamic dataflow architecture using partial reconfigurable hardware as an option for multiple cores *W. Trans. on Comp*, WSEAS, v.9, n.5, pp.429-444.

[15] Teifel, J. Rajit, M.(2004) An asynchronous dataflow FPGA architecture, *IEEE Transactions on Computers*, IEEE, pp. 1376–1392.

[16] Bobda, C. (2007) *Introduction to Reconfigurable Computing* , Springer.

[17] Xilinx, (2011) *Xilinx XST User Guide for Virtex-6, Spartan-6, and 7 Series Devices*, Xilinx.

[18] K. Papadimitriou, A. Dollas, S. Hauck. Performance of Partial Reconfiguration in FPGA Systems: A Survey and a Cost Model, *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, ACM, USA, to appear.

[19] Mingw (2009) *Mingw.org*, Mingw - minimalist gnu for windows.

[20] Microsoft (2010) *Queryperformancecounter function*, Microsoft Corporation.

[21] Intel (1998) *Using the rdtsc instruction for performance monitoring*, Intel Corporation.