A Dynamic Dataflow Architecture using Partial Reconfigurable Hardware as an option for Multiple Cores

JORGE LUIZ E SILVA
University of Sao Paulo
Department of Computer Systems
Av. Trabalhador Saocarlense, 400
BRAZIL
jsilva@icmc.usp.br

JOELMIR JOSÉ LOPES
University of Sao Paulo
Department of Computer Systems
Av. Trabalhador Saocarlense, 400
BRAZIL
joelmir@icmc.usp.br

Abstract: Different from traditional processors, Moore´s Law was one of the reasons to duplicate cores, and at least until today it is the solution for safe consumption and operation of systems using millions of transistors. In terms of software, parallelism will be a tendency over the coming years. One of the challenges is to create tools for programmers who use HLL *(High Level Language)* producing hardware directly. These tools should use the utmost experience of the programmers and the flexibility of FPGA *(Field Programmable Gate Array)*. The main aspect of the existing tools which directly convert HLL into hardware is dependence graphics. On the other hand, a dynamic dataflow architecture has implicit parallelism. ChipCflow is a tool to convert C directly into hardware that uses FPGA as a partial reconfiguration based on a dynamic dataflow architecture. In this paper, the relation between traditional dataflow architecture and contemporary architecture, as well as the main characteristics of the ChipCflow project will be presented.

Key–Words: Dataflow Architecture; Reconfigurable Hardware; Tagged-token; Run-time Reconfiguration; Protocol for dataflow.

1 Introduction

We present a FPGA dataflow architecture to implement high-performance logic. Dataflow machines are programmable computers in which the hardware is optimized for fine-grain data-driven parallel computation.

There is no sharp definition of dataflow machines in the sense of a widely accepted set of criteria to distinguish dataflow machines from all other computers, [36] considering that dataflow machines are all programmable computers where the hardware is optimized for fine-grain data-driven parallel computation [37]. Fine grain means that the processes that run in parallel are approximately of the size of a conventional machine code instruction [36], [37].

Data driven means that the activation of a process is solely determined by the availability of its input data. This definition excludes simulators, as well as nonprogrammable machines, for instance those that implement the dataflow graph directly in the hardware which is an approach that is popular in constructing dedicated signal processors [36], [18].

The dataflow model of computation offers a simple, yet powerful, formalism for describing parallel computation and this systems represents a unique class of computer architecture which combines a heterogeneous, fine-grain model of computation with latency hiding mechanisms. In contrast to the von Neumann model of computation, the execution of an instruction in the Dataflow model relies on the availability of its operands, rather than on a predefined sequence of instructions [2], [30]. Even in parallel versions of the von Neumann model, sequencing of instructions is controlled explicitly by the programmer or compiler. In a Dataflow system, the selection of instructions for execution is performed using the hardware at execution time and is constrained only by the partial order implicit in the program's data dependency graph. The result of the computation is fine-grained and shows a much higher degree of parallelism than codes written for parallel von Neumann machines. This fine-grained parallelism is then used for exploiting replicated hardware for increased performance, masking memory access latency, and maintaining a uniform distribution of workload [30].

This kind of architecture was first researched in the 1970s and was discontinued in the 1990s ([5]; [11]; [14]). With the advance of technology of microelectronics, the Field Programable Gate Array (FPGA) has been used, mainly because of its flexibility, the facilities to implement complex systems and intrinsic parallelism [29].

The datafow architecture is a topic which has come to light again [9]; [33], especially because of the reconfigurable architecture, which is totally based on FPGAs. Static and dynamic dataflow architectures are presented as two implementations of the abstract dataflow model [2]. In particular, this paper will discuss the dynamic architecture, which is based on the architecture generated by Chipcflow.

System designers have been finding the cost/performance trade-offs tipping increasingly in favor of FPGA devices over high-performance DSP *(Digital Signal Processor)* chips and perhaps most significantly when compared to the risks and up-front costs of a custom ASIC solution. Combining the flexibility of the GPP *(General Purpose Processor)* and the efficiency of ASIC *(Application-Specific Integrated Circuit)* in one device has been proven to be a good solution [26].

Most computationally intensive algorithms can be described using a relatively small amount of C source code when compared to a hardware-level, equivalent description. The ability to quickly try out new algorithmic approaches from C-language models is therefore an important benefit of using a software-oriented approach to design. Reengineering low-level hardware designs, on the other hand, can be a tedious and error-prone process [26], due to the need of a solid background in logic and circuit design. Furthermore, the programming tools chain is long and complex when compared to the simple compilation step of traditional languages [23].

When a program is written in a high level language like C and C++, an equivalent workable and easy to modify code for a given reconfigurable system should be automatically generated. The user will program a reconfigurable architecture without having to deal with issues like hardware/software partitioning, task distribution, simulation, timing analysis and hardware reconfiguration. The system should do the job for the user [7].

As FPGAs have grown in logic capacity, their ability to host high-performance software algorithms and complete applications has grown correspondingly [26], [8], [16]. For software engineers, the main aim is to present FPGAs as software-programmable computing resources.

Software development methods and software languages can be used in a practical way to create FPGA-based, high-performance computing applications, without a deep knowledge of hardware design. Even when the entire hardware design is eventually recorded with a lower-level HDL, high-level design languages enable hardware engineers to rapidly explore the design space and create working prototypes.

The main advantages in using a C compiler to FPGA is the approach to hardware abstraction where the programmer does not need to understand all the details of the hardware target, and yet is guided by the programming model, towards more appropriate methods of coding and a balance between software design productivity and hardware design results, as measured in system performance and size can be achieved.

While FPGA-design time remains drastically shorter then ASIC-design time, implementing a function in FPGA can still take days, weeks, or even months. This is not acceptable for a software programmer or a mechanical engineer, who is used to implementing applications on a general purpose computer in a few minutes or hours with far less difficulty and knowledge than required by FPGA programming tools [7].

The Chipcflow project is a system where a C program is initially converted into a Dynamic Dataflow graph, followed by its execution in Reconfigurable Hardware. A dynamic partial reconfiguration, present in some FPGAs is explored and provides dynamic dataflow execution.

This paper describes the architecture of the hardware generated by the Chipcflow. The architecture generated by the tool explores one of the main problems encountered in dataflow research: the management of data structures but more specifically arrays. Given that the semantics of Data-Flow languages are basically functional in nature, the modification of a single element of an array needs the creation of another array, identical to the original, except for the altered element. Multiple references to an array require multiple copies of the array, even when only one element is needed.

The remainder of the paper is organized as follows: section 2 focuses on dataflow model of execution, related architectures and the tagged-token model. Section 3 emphasizes on related work. Section 4 shows the basic structure for Chipcflow: the compiler, its operators, and some examples of graphs which are presented. Iterative constructors are described which enable various instances of an operator to be executed in the dynamic model of dataflow using an iterative constructor respectively. The Matching data that identifies items of data partners is described. The implementation of the operator and its instances are also described and some details of implementation are shown. Finally, the management of data structures and control iterative constructors are presented, specifically how these structures are dealt with, and their differences with the previous ones.

2 Dataflow Model

The dataflow model of execution offers attractive properties for parallel processing. First, it is asynchronous: due to the fact that it bases instruction execution on operand availability, synchronization of parallel activities is implicit in the dataflow model. Second, it is self-scheduling: except for data dependencies in the program, dataflow instructions do not constrain sequencing; hence, the dataflow graph representation of a program exposes all forms of parallelism, eliminating the need to explicitly manage parallel execution. For high-speed computations, the advantage of the dataflow approach over the control-flow method stems from the inherent parallelism embedded at the instruction level. This enables efficient exploitation of fine-grain parallelism in application programs [20].

In the most recent data-flow machines are multiprocessors which execute parallel program graphs rather than sequential programs. The order of execution of the nodes in the graph (or instructions) is determined by the availability of their operands rather than the strict sequencing of instructions in a von Neumann machine. Consequently the program statements are executed in a non-deterministic manner, and parallelism is obtained if more than one node executes at the same time. Figure 1 shows a sample dataflow graph for an arithmetic expression and Figure 2 shows a model for the hardware required to execute such data-flow programs. In this hardware, the program graph is distributed to the processing elements (PEs) so that the computation of A*B can proceed at the same time as C+D.

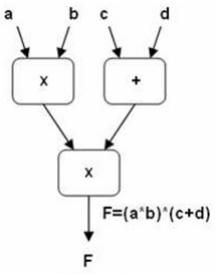

Figure 1: A data-flow graph

In Data-Flow systems, data values rather than being stored at particular addresses are tagged. The tag includes the address of the instruction for which the particular data value is destined and other information defining the computational context in which that value is used. This context is called the value's color. The data value, together with its tag, is called a token [30].

A dataflow program is described by a directed graph where the nodes denote operations, e.g. addition and multiplication, and the arcs denote data dependencies between operations, Figure 1 [2].

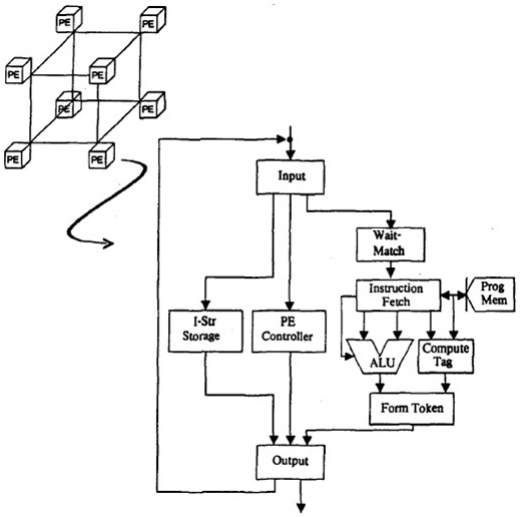

Figure 2: The processing element (PE) of the MIT tagged token dataflow machine, [2]

Any arithmetic or logical expression can be translated into an acyclic dataflow graph in a straightforward manner, Figure 1. Data values are carried on tokens, which flow along the arcs. A node may execute (or fire) when a token is available on each input arc. When it fires, a data token is removed from each input arc, a result is computed using these data values, and a token containing the result is produced on each output arc [2].

In order for an instruction, requiring two operands to execute, both tokens must exist and be brought together. This synchronization process is called matching. Once these input tokens are matched, the instruction is performed, and the result token(s) are sent on to subsequent instructions. Note that tokens which do not require matching may go directly to the execution unit. These tokens are called by-pass tokens.

Dataflow graphs can be viewed as a machine language for a parallel machine where a node in a dataflow graph represents a machine instruction. The instruction format for a dataflow machine is essentially an adjacency list representation of the program graph: Each instruction contains an op-code and a list of destination instruction addresses. Recall that an instruction or node may execute whenever a token is available on each of its input arcs and that when it fires, the input tokens are consumed, result value is computed and a result token is produced on each output arc. This dictates the following basic instruction cycle: (a) detect when an operation is enabled (this is equivalent to collecting operand values); (b) determine the operation to be performed, i.e. fetch the instruction; (c) compute results; and (d) gener-

ate result tokens. This is the basic instruction cycle of any dataflow machine; however, there remains tremendous flexibility in the details of how this cycle is performed.

It is interesting to contrast dataflow instructions with those of conventional machines. In a von Neumann machine, instructions specify the addresses of the operands explicitly and the next instructions implicitly via the program counter (except for branch instructions). In a dataflow machine, operands (tokens) carry the address of the instruction for which they are destined, and instructions contain the addresses of the destination instructions. Since the execution of an instruction is dependent upon the arrival of operands, instruction scheduling and management token storage are closely related in any dataflow computer.

Dataflow graphs show two kinds of parallelism in instruction execution, [2]:

- The first we might call spatial parallelism: Any two nodes can potentially execute concurrently if there is no data dependence between them;
- The second form of parallelism results from pipelining independent waves of computation through the graph.

The essential point to keep in mind in considering ways to implement the dataflow model is that tokens imply storage. The token storage mechanism is the key feature of dataflow architecture. There are currently two main classifications for dataflow architectures, static and dynamic or Tagged-Token Dataflow.

In the static dataflow model only one token (or instruction operand) is allowed on a program arc at any time. In the dynamic model many tokens are allowed on arcs, and their order is determined by special tag fields. A good overview of these architectures can be found in [31]. The static scheme was first proposed by Dennis [12],[13],[14].

In the abstract dataflow model, data values are carried on tokens, which travel along the arcs connecting various instructions in the program graph, and it is assumed that the arcs are first-in-first-out (FIFO) queues of unbounded capacity [20].

The tagged-token approach eliminates the need to maintain FIFO queues on the arcs, as in the static dataflow model, (though unbounded storage is still assumed) and consequently offers more parallelism than the static model [2]. The dynamic scheme is used in Arvind's research group in MIT [3],[4] and in Manchester University.

2.1 TAGGED-TOKEN DATAFLOW MACHINE

Dataflow architectures can also be classified as centralized or distributed, based on the organization of their instruction memories.

The dynamic dataflow organization from MIT is a multiprocessor system in which the instruction memory is distributed among the processing elements. The choice between centralized or distributed memory organizations has a direct effect on program allocation [20]

The Tagged-token dataflow machine proposed by Arvind et al ([6]) is depicted in Figure 2. It comprises a collection of PEs connected via a packet communication network. Each PE is a complete dataflow computer. The waiting-matching store is a key component of this architecture. When a token enters the waiting-matching stage, its tag is compared against the tags of the tokens resident in the store. If a match is found, the matched token is purged from the store and is forwarded to the instruction fetch stage, along with the entering token. Otherwise, the incoming token is added to the matching store to await its partner. (Instructions are restricted at most to two operands, so a single match enables an activity.) Tokens that require no partner, i.e. are destined for a monadic operator and bypass the waiting-matching stage [2].

The detection of matching tokens is one of the most important aspects of the dynamic dataflow computation model [20]. Once an activity is enabled, it is processed in a pipelined fashion without further delay.

Tags of the MIT tagged token dataflow machine [2] have four parts: invocation ID, iteration ID, code block, and instruction address. The latter two identify the destination instruction and the former two identify a particular firing of that instruction. The iteration ID distinguishes between different iterations of a particular invocation of a loop code-block, while the invocation ID distinguishes between different invocations.

The invocation ID in the tag designates a set of three registers (CBR, DBR, and MAP) that contain all the information associated with the invocation. CBR contains the base address of the code block in program memory; DBR contains the base address of a data area that holds values of loop variables that behave as constants, and MAP contains mapping information describing how activities of the invocation are to be distributed over a collection of PEs. The instruction fetch stage is thus able to locate the instruction and any required constants. The op-code and data-values are passed to the arithmetic logic unit (ALU) for processing. In parallel with the ALU, the computed tag stage accesses the destination list of the instruction and prepares result tags using the mapping

information. Result values and tags are merged into tokens and passed to the network, where upon they are routed to the appropriate waiting-matching store [2]

Therefore, in addition to the functional units described in Figure 2, each PE must have a token buffer. This buffer can be placed at a variety of points, including the output stage or the input stage, depending on the relative speeds of the various stages. Both the waiting-matching store and the token buffer have to be large enough to make the probability of overflow acceptably small [2].

The invocation request is passed to a system-wide resource manager so that resources such as a new invocation ID, program memory etc, can be allocated for the new invocation.

A code-block invocation can be placed on essentially any collection of processors. Various instances, i.e. firings, of instructions are assigned for PEs within a collection by "hashing" the tags [2].

The management of data structures, arrays in particular, is one of the major problems in Data-Flow research. Given that the semantics of Data-Flow languages are basically functional in nature, the modification of a single element of an array needs the creation of another array, identical to the original, except for the altered element. Multiple references to an array require multiple copies of the array, even when only one element is needed. Solutions are varied and depend largely on the architecture [30].

Data structures have two modes of reference, to the data structure as a whole and to the individual elements.

Data structures in Data-Flow systems require special treatment. In most systems, there is an additional, specialized function unit (e.g. the structure store in the Manchester machine [2] which provides the storage and performs token colouring.

Another disadvantage, specially for the dynamic model is that data may be output in any order and must be resorted to if ordering is important. This may add a considerable overhead to the computation [1].

Thus in cases when pipelining is the most important form of parallelism and there is little loop unfolding or feedback, as in many real world control problems, the dynamic architectures have the added complexities of tagging and untagging, the increased network traffic and the resorting of data. For example, each iteration of a loop in a dynamic machine must include a tag generation code, even if the loop has a data dependency in it that forbids loop unfolding. This overhead is not present in the static model [1].

3 Related work

More recently, new dataflow architectures were proposed, mainly TRIPS and WaveScalar [34], [22], [32], [28], [21], [35]. These architectures are based in an Tiled architectures. The basic premise of these architectures is that larger, higher-performance implementations can be constructed by replicating the basic tile across the chip.

Many computer architects are beginning to shift their focus away from today's complex, monolithic, high-performance processors. Instead, they are designing a much simpler processing element (PE) and compensating for its lower individual performance by replicating it across a chip. PE replication provides robustness in the face of fabrication errors, and the combination reduces wire delay for both data and control signal transmission. The result is an easily scalable architecture that enables a chip designer to capitalize on future silicon process technologies [34].

WaveScalar is a tagged-token, dynamic dataflow architecture. Like all dataflow architectures its application binary is a program dataflow graph. Each node in the graph is a single instruction which computes a value and sends it to the instructions that consume it. An instruction executes after all its input operand values arrive according to a principle known as the dataflow firing rule. WaveScalar can execute programs written with conventional von Neumann-style memory semantics (i.e. those composed in languages like C/C++) and correctly orders memory operations with a technique called wave-ordered memory [5]. A PE contains all the logic for dataflow execution. It has an input interface that receives tokens containing a data value and information that associates the value with a particular instruction. These tokens are stored in a matching table, which is implemented as a small, non-associative cache [34].

The TRIPS / GPA [22] processor is a recent VLIW-dataflow hybrid. TRIPS bundles hyperblocks of VLIW instructions together vertically and describes their dependencies explicitly instead of implicitly through registers. Within a hyperblock, instructions fire according to the dataflow firing rule, while communication between hyperblocks occurs through a register file. At its core, TRIPS remains a von Neumann architecture, because the program counter still determines the sequence of hyperblocks the processor executes. TRIPS is an innovative way to build a Very Long Instruction Word (VLIW) processor from next generation silicon technology. A VLIW bundles instructions horizontally to be executed in parallel [32].

The TRIPS [22]; [28] processor uses dataflow ideas to build a hybrid von Neumann/dataflow ma-

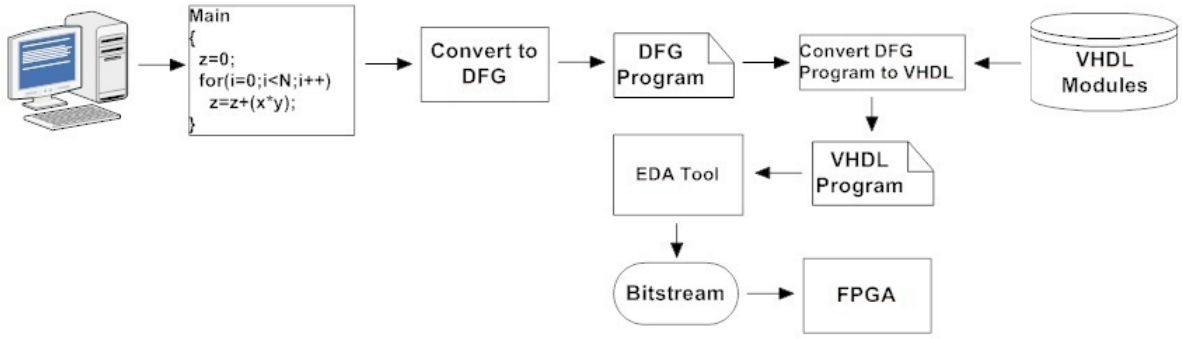

Figure 3: The Flow Diagram for Chipcflow tool

chine. It uses a program counter to guide execution, but instead of moving from one instruction to the next, the TRIPS PC selects frames (similar to hyperblocks [21]) of instructions to execute in an array of 16 processing elements that make up a TRIPS processor [35].

The TRIPS processor and WaveScalar share the same technological challenges, and tend to use the same terminology to describe aspects of their designs. However, the only architectural feature TRIPS and WaveScalar share is the use of direct links between instructions of the same hyper-block (in TRIPS) directed acyclic graph (in WaveScalar).

Despite high-level similarities between waves and frames and the WaveScalar and TRIPS PE designs, the two architectures are quite different. Because it uses a program counter to select frames of instructions for execution, TRIPS must speculate aggressively. Mapping a frame of instructions onto the PE array takes several cycles, so the TRIPS processor speculatively maps frames onto the PEs ahead of time. WaveScalar does not suffer from this problem because its dynamic dataflow execution model enables the instructions to remain in the grid for many executions, making the need for speculation obvious. The disadvantage of the WaveScalar's approach is the need for complex tag-matching hardware to support dynamic dataflow execution [35]

The two projects also have much in common. Both take a hybrid static/dynamic approach to scheduling instruction execution by carefully placing instructions in an array of processing elements and then allowing execution to proceed dynamically. This places both architectures between dynamic out-of-order superscalar designs and statically scheduled VLIW machines. These designs have run into problems because dynamic scheduling hardware does not scale and by nature, static scheduling is conservative. A hybrid approach will be necessary, but it is as yet unclear whether either WaveScalar or TRIPS strikes the optimal balance [35].

FPGAs can also be viewed as tiled architectures and can offer insight into the difficulties tiled processor designers may face. FPGAs already provide heterogeneous arrays of tiles (e.g., simple lookup tables, multipliers, memories, and even small RISC cores) and vary the mix of tiles depending on the size of the array [34].

4 The Chipcflow tool

The ChipCflow project is a system where a C program is initially converted into a Dynamic Dataflow graph, followed by its execution in Reconfigurable Hardware. Its flow diagram is shown in Figure 3. The ChipCflow system begins in a host machine where a C program is used to be converted into a control dataflow graph (CDFG [1]) generating a CDFG object program. The CDFG program is converted into a VHDL where modules of CDFG are accessed from a data base of VHDL modules, where there are all operators of Chipcflow implemented in VHDL. After generating the complete VHDL program, an EDA tool to convert the VHDL program into a bitstream and to download it to a FPGA is used.

4.1 The Operators and C statements implemented in Dataflow Graphs

The operators to be used in the Chipcflow project are: "decider", "non deterministic merge", "determin-

[1]The CDFG is a directed acyclic graph in which a node can be either an operation node or a control node (representing a branch, loop, etc.) [24]. The directed edges in a CDFG represent the transfer of a value or control from one node to another. An edge can be conditional representing a condition while implementing the if/case statements or loop constructs. The CDFG has the advantage of depicting both control and data constructs in a single graph, giving a better state-space exploration capability.

istic merge", "branch", "copy" and "operator". They are described in Figure 4 and Figure 5.

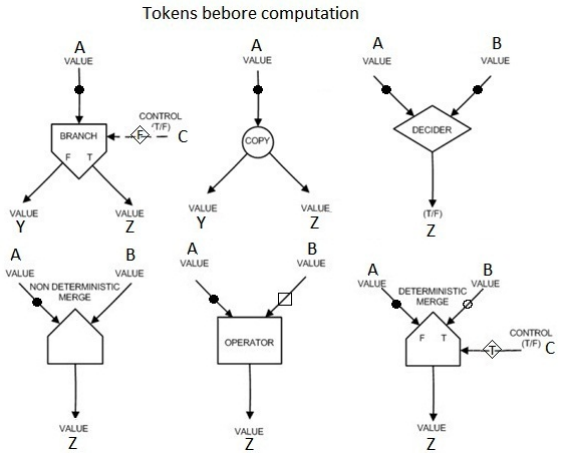

Figure 4: Dataflow control and computation nodes, tokens arrive at operators, (filled circles, empty circles, empty square and empty triangle indicate different tokens of arrival to the operators).

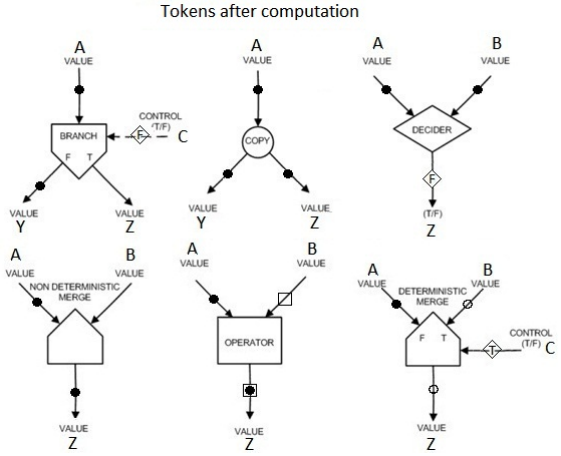

Figure 5: Dataflow control and computation nodes, tokens fired.

Using these dataflow nodes, we can build any deterministic dataflow graph. These dataflow nodes behave as follows:

1. Copy: This dataflow node duplicates tokens to two receivers. It receives a token on its input channel and copies the token to all of its output channels.

2. Function: This dataflow node computes arbitrary functions of two variables. These functions can be "plus", "multiplier", etc. It waits until tokens have been received on all its input channels and then generates a token with results on its output channel.

3. Deterministic merge: it receives a control token in channel C. If the control token has a FALSE value, it reads a data token from channel A, otherwise it reads a data token from channel B. Finally, the data token is sent to channel Z. A merge node is similar to a multiplexer except that a token on the unused conditional input channel will not be consumed and does not need to be present for the merge node to process tokens on the active input data channel.

4. Branch: This dataflow node performs a two-way controlled token branch and allows tokens to be conditionally sent to channels. It receives a control token in channel C and a data token in channel A. If the control token has a FALSE value, it sends the data token to channel Y; otherwise it sends the data token to channel Z. A branch node is similar to a demultiplexer, except that no token is generated in the unused conditional output channel in its implementation.

5. The Decider operator will be used to generate a control signal "TRUE" or "FALSE" after executing a boolean operation such as "¿", "¡=", "/=", etc.

6. The Non-deterministic merge operate like a deterministic merge, except that there is no control token. This operator acts as first-in, first-out. For example, if a data token comes first in channel A, this token will be sent to output Z, otherwise a token in channel B will be sent to output Z.

In order to build conditional and loop program graphs, there are two control operators: branch and merge. Unlike the other operator in Chipcflow, the branch and merge are not well-behaved in isolation, but yield well-behaved graphs when used in conditional and loop schemes [2]. Each graph is either acyclic or a single loop.

The dataflow graph of the While statement was implemented using these operators and is described in Figure 6. In this figure, there are two branch operators; two deterministic merges; five copy operators; one decider with a boolean operator "¿" and two operators with an arithmetic operation "+".

In the next section, the basic structure for the C compiler and some examples of graphs are presented.

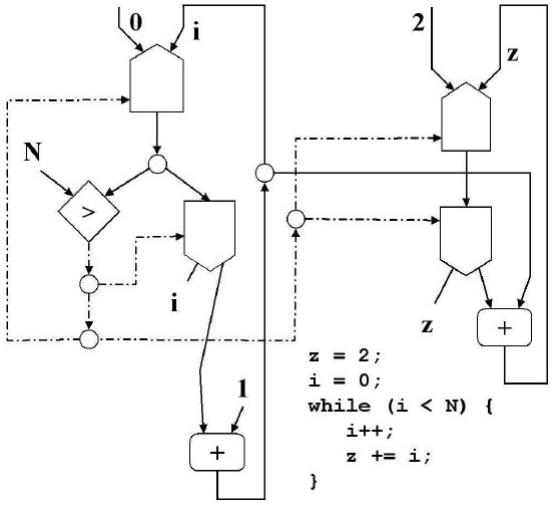

Figure 6: Example of graph extracted from a while command

4.2 The C Compiler to generate dataflow graph

The compiler was designed and developed to generate a control and dataflow graph (CDFG) from a source code written in C language. The compiler structure was implemented in C++ and it is made up of two main parts: the CDFG extraction and VHDL code generator [29].

The compiler generates a series of intermediate files, such as a binary mapping file that is then used to mount the CDFG and then the compiler generates the VHDL that can be synthesized by a tool such as Xilinx ISE.

4.2.1 Generating a Binary Mapping File after Lexical Analysis

To generate a Binary Mapping file, a token was defined and its format is described in Figure 7. The first 4-bits of the token were used to identify the operator; the second, the third and the fourth 5-bits were used to identify the three inputs (a, b and c) of the operator; finally, the sixth and the seventh 5-bits to identify the outputs (s and z) of the operator. This is a generic template for the operator with three inputs and two outputs signal, however there are operators with less than three input signals and just one output signal.

Operator	Input a	Input b	Input c	output s	output z

Figure 7: The format of the token

The packet of bits for this particular operator can be clearly seen in the first packet of bits in Figure 8, which is in accordance with the format described in Figure 7. The "xxxxx" in the packet of bits represent an arc with no connection signal. Thus, a file with these packets of bits is a binary representation for a dataflow graph extracted from a while C statement in the C compiler [29].

```
00010000000001000100011XXXXX
00011000110010010101010 1XXXXX
00100010100111XXXXX0110101110
00111011000100XXXXX00110XXXXX
10000111001111XXXXX10000XXXXX
```

Figure 8: the Binary Mapping File generated for While C command.

The next step is to use the binary representation to identify the VHDL operators, which components will be used and which instances and their interconnections will be generated to execute the VHDL program in the ISE Xilinx platform.

In order to generate a VHDL program, the file with the binary mapping is converted using the operators in VHDL Modules shown in Figure 3, which is already implemented in VHDL having their instances and interconnections.

A complete example of the process is described below. Algorithm 1 is used to generate the CDFG graph and the corresponding VHDL code. In Figure 9, a C program to be converted into a VHDL and correspondent dataflow graph are described.

As discussed above, the compiler is made up of two main parts: the CDFG extraction and VHDL code generator. After the CDFG extraction, the correspondent dataflow graph representation is shown in Figure 9, which is a graphic representation of the dataflow graph described in Figure 9.

In the VHDL code generator phase, the compiler uses binary mapping and the modules are selected from a library which contains the Chipcflow operators, already implemented in VHDL and stored in a VHDL module data base, Figure 3. The final result is a VHDL structural description file which has the Chipcflow operators placed and connected together in the CDFG graph as generated by the compiler. Finally, the VHDL files are ready to be executed in the tool as the ISE Foundation from Xilinx [29]. In the listing 1, the VHDL code generated for the binary mapping is described in the Figure 9.

Listing 1: The *IF, FOR, WHILE* Command in

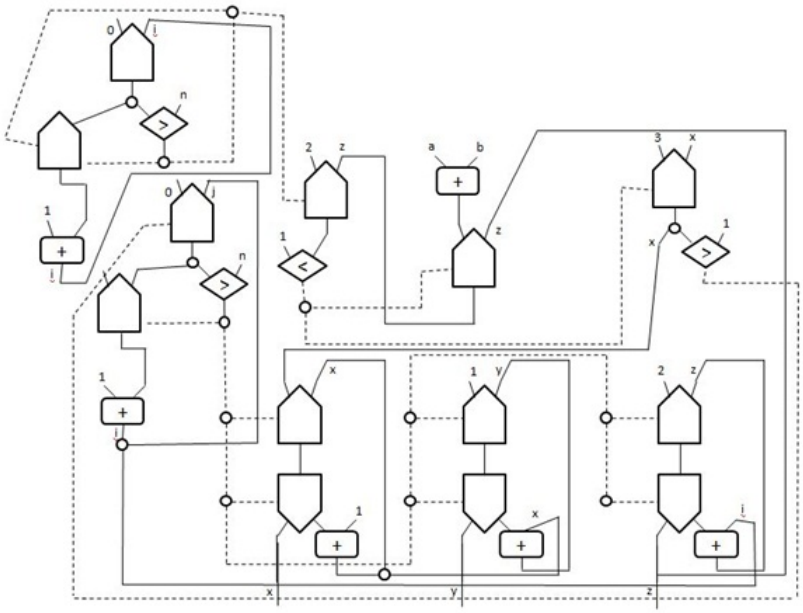

Figure 9: the Example of C Program and correspondent Dataflow Graph

Algorithm 1 Code with $IF, FOR, WHILE$ Commands

$a \Leftarrow 3$
$b \Leftarrow 0$
$y \Leftarrow 1$
$z \Leftarrow 2$
for $I = 0$ to n **do**
  if $z > 1$ **then**
    $z \Leftarrow a + b$
  else
    $x \Leftarrow 3$
    while $x > 1$ **do**
      for $j = 0$ to $j < n$ **do**
        $x \Leftarrow x + 1$
        $y \Leftarrow y + x$
        $z \Leftarrow z + i$
      end for
    end while
  end if
end for

VHDL[29]

```
1.  ----------------------------------------------------------------------------------
2.  -- Company:  Projeto ChipCflow
3.  -- Engineer:  kelton Augusto Pontara da Costa
4.  -- Create Date:  08:43:20 03/27/2009
5.  -- Module Name:  Circuito_completo - Behavioral
6.  -- Revision 0.01 - File Created
7.  ----------------------------------------------------------------------------------
8.  library ieee;
9.  use ieee.std_logic_1164.all;
10. use ieee.std_logic_arith.all;
11. use ieee.std_logic_signed.all;
12.
13. --- uncomment the following library declaration
14.       if instantiating
15. --- any xilinx primitives in this code
16. --library UNISIM
17. --use UNISIM.VComponents.all
18.
19. ENTITY circuito_final IS
20. port( a : in std_logic_vector(31 downto 0);
21. b : out std_logic_vector(31 downto 0);
22. END;
23.
24. ARCHITECTURE behavior OF circuito_final IS
25.
26. Component merge_deterministic
27. port(
28. a: in std_logic_vector(31 downto 0);
29. b: in std_logic_vector(31 downto 0);
30. c: out std_logic_vector(31 downto 0);
31. z: out std_logic_vector(31 downto 0)
32. );
33. END Component;
34.
35. Component copy
36. port(
37. a: in std_logic_vector(31 downto 0);
38. z: out std_logic_vector(31 downto 0);
39. y: out std_logic_vector(31 downto 0)
40. );
41. END Component;
42.
43. Component decider
44. port(
45. a: in std_logic_vector(31 downto 0);
46. b: in std_logic_vector(31 downto 0);
47. z: out std_logic_vector(31 downto 0)
48. );
49. END Component;
50.
51. Component operator
52. port(a: in std_logic_vector(31 downto 0);
53. b: in std_logic_vector(31 downto 0);
54. z: out std_logic_vector(31 downto 0)
55. );
```

```
56.     END Component;
57.
58.     Component branch
59.     port(
60.     a: in std_logic_vector(31 downto 0);
61.     c: in std_logic_vector(31 downto 0);
62.     z: out std_logic_vector(31 downto 0);
63.     y: out std_logic_vector(31 downto 0)
64.     );
65.     END Component;
66.
67.     signal i10, i36, i12, i30, i15, i33, i18, i37, i20,
68.     i38, i39, i40, i11, i13, i14, i41, i16, i17, i19, i42,
69.     i21, i26, i22, i23, i24, i25, i27, i28, i29, i31, i32,
70.     i43, i34, i35, i44, i111, i112, i131, i132, i171, i172,
71.     i211, i212, i221, i222, i231, i232, i231, i232,
72.     i231, i232, i231, i232, i231, i232, i231, i232, i251,
73.     i252, i291, i292 : std_logic_vector(31 downto 0);
74.
75.     BEGIN
76.     u1: merge_deterministic port map (i10,i36,i11);
77.     u2: copy port map (i11,i111,i112);
78.     u3: decider port map (i12,i111,i13);
79.     u4: copy port map (i13,i131,i132);
80.     u5: merge_deterministic port map (i112,i131,i14);
81.     u6: operator port map (i14,i30,i41);
82.     u7: merge_deterministic port map (i15,i33,i132,i16);
83.     u8: decider port map (i16,i30,i17);
84.     u9: copy port map (i17,i171,i172);
85.     u10: operator port map (i18,i37,i19);
86.     u11: merge_deterministic port map (i19,i15,i171,i42);
87.     u12: merge_deterministic port map (i20,i38,i172,i21);
88.     u13: copy port map (i21,i211,i212);
89.     u14: decider port map (i211,i30,i26);
90.     u15: merge_deterministic port map (i10,i39,i26,i22);
91.     u16: copy port map (i22,i221,i222);
92.     u17: decider port map (i221,i12,i23);
93.     u18: copy port map (i23,i231,i232);
94.     u19: copy port map (i23,i231,i232);
95.     u20: copy port map (i23,i231,i232);
96.     u21: copy port map (i23,i231,i232);
97.     u22: copy port map (i23,i231,i232);
98.     u23: copy port map (i23,i231,i232);
99.     u24: merge_deterministic port map (i222,i231,i24);
100.    u25: operator port map (i24,i30,i25);
101.    u26: copy port map (i25,i251,i252);
102.    u27: merge_deterministic port map (i26,i38,i232,i27);
103.    u28: branch port map (i27,i233,i28,i28);
104.    u29: operator port map (i28,i30,i29);
105.    u30: copy port map (i29,i291,i292);
106.    u31: merge_deterministic port map (i30,i40,i234,i31);
107.    u32: branch port map (i31,i235,i32,i32);
108.    u33: operator port map (i32,i291,i43);
109.    u34: merge_deterministic port map (i33,i15,i236,i34);
110.    u35: branch port map (i34,i237,i35,i35);
111.    u36: operator port map (i35,i251,i44);
112.    END behavior;
```

4.3 The system generated by Chipcflow

In most dataflow machines, the hardware is implemented as a collection of PEs fixed in the hardware and the compiler using sophisticated optimization techniques is able to distribute the application of users in these PEs. [36]; [10]; [17]; [25]; [19]; [27]; [15]; [34].

The Chipcflow architecture is different; the architecture is only generated based on the user application. The Chipcflow architecture focuses on solving one of the major research problems in dataflow, the management of data structures, particularly arrays as already discussed above and still optimized to save space and power due to exploring technology of partial and dynamic reconfigurations in the Virtex family FPGA from Xilinx.

Figure 10 shows the Chipcflow token. The fields (Activation, Nesting and Iteration) make up the tag of the token. The field ”data” represents the data value that the tokens carry.

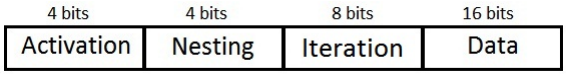

Figure 10: the format of the Tagged-Token

4.3.1 Matching unit

The matching unit is a key component of Chipcflow architecture. Its main objective is to synchronize the token input in the operators. Once all the input tokens of the operators are matched, the instruction is performed, and the result token(s) are sent on to subsequent instructions. The description of this unit has been discussed above.

The matching unit with 3 inputs can be seen in Figure 11.

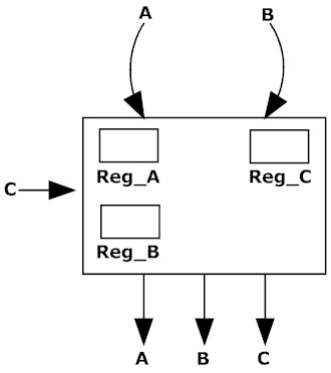

Figure 11: The matching unit with 3 inputs

As can be seen in Figure 11, this unit has input tokens, internal registers to the inputs and output tokens.

The Design Statistics for Matching Unit with 2 inputs can be observed in Table 1.

The design statistics for the matching unit with 3 inputs can be observed in Table 2.

When a token arrives at the input, a series of tests are made with their tags: if a token arrives in input A, for example, the corresponding registers of the other entries are tested to verify if they are empty. If true, the token of input A is stored in register A. On the other hand, if register B and / or C are not empty, the token in input A is compared with them. If they are equal, then tokens A, B and C are sent to the output of the matching unit and therefore processed by the operator. Otherwise an instance of this operator is created and the token is stored in register A of the matching unit belonging to this operator. Instances of operators are described in the following sections.

Operands that have matching unit are: the branch, operator and deterministic-merge. On the other hand, tokens which do not require matching may go directly

Table 1: Matching unit with 2 inputs using device 2vp30ff896-6 of Xilinx FPGA

Device Elements	Range	Utilization
Number of Slice Flip Flops	64 out of 27.392	0%
Number of 4 Input LUTs	80 out of 27.392	0%
Number of Slices	46 out of 13.696	0%
Numbers of IOs	120	
Number of bonded IOBs	120 out of 556	21%
Number of GCLKs	2 out of 16	12%
Minimum period: 2.228ns (Maximum Frquency: 448.827Mhz		
Minimum input arrival time before clock: 5.780ns		
Maximum output required time after clock: 4.881ns		
Maximum combinational path delay: 5.549ns		

Table 2: Matching unit with 3 inputs using device 2vp30ff896-6 of Xilinx FPGA

Device Elements	Range	Utilization
Number of Slice Flip Flops	96 out of 27.392	0%
Number of 4 Input LUTs	78 out of 27.392	0%
Number of Slices	178 out of 13.696	0%
Numbers of IOs	149	
Number of bonded IOBs	149 out of 556	26%
Number of GCLKs	3 out of 16	18%
Minimum period: 3.038ns (Maximum Frquency: 329.164Mhz		
Minimum input arrival time before clock: 6.535ns		
Maximum output required time after clock: 5.873ns		
Maximum combinational path delay: 6.758ns		

to the output of operators. These tokens are called by-pass tokens. Operators that do not contain matching units are: the copy and non-deterministic-merge. The Chipcflow architecture matching units may contain two or three inputs, depending on the number of entries in the operator.

4.3.2 The branch operator

This section shows the implementation of the branch operator, used by Chipcflow. This operator is shown in Figure 12. The results of the implementation are described in Table 3.

The Branch operator consists of control units. These units have specific functions in the operator as follows:

- Matching unit: this is a matching unit with 3 inputs, as discussed above. When all the tokens in the input of the operator are equal, the matching unit sends it to the multiplexer unit. Otherwise your inputs are sent to the place unit.

- Multiplex unit: this unit is similar to a multiplexer. The output of this unit is sent to the fire unit. The understanding of the multiplexer will be explained in the next item.

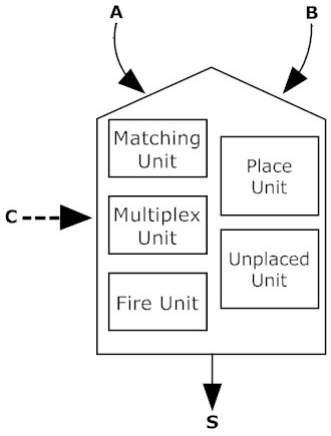

Figure 12: Branch unit operator

- Place unit: this unit is responsible for the protocol to create new instances of the operator. The model of creating new instances will also be shown in the next item.

- Unplaced unit: this unit is responsible for real-

Table 3: Branch unit operator, using the Virtex device from Xilinx 2vp30ff896-6

Device Elements	Range	Utilization
Number of Slice Flip Flops	132 out of 27.392	0%
Number of 4 Input LUTs	106 out of 27.392	0%
Number of Slices	240 out of 13.696	0%
Numbers of IOs	180	
Number of bonded IOBs	180 out of 556	32%
Number of GCLKs	4 out of 16	25%
Minimum period: 3.038ns (Maximum Frquency: 329.164Mhz		
Minimum input arrival time before clock: 6.558ns		
Maximum output required time after clock: 5.880ns		
Maximum combinational path delay: 6.765ns		

locating instances of the operator to maximize available space to allocate new ones.

- Fire unit: this unit is responsible for sending the result token to the subsequent operator.

4.3.3 The model to create new instances

The Branch operator and deterministic-merge operators have a matching unit, place unit and unplaced unit, as shown in Figure 12. Each of these operators has the ability to allocate new instances. The instances are allocated or deallocated when all partner tokens of an instance are available.

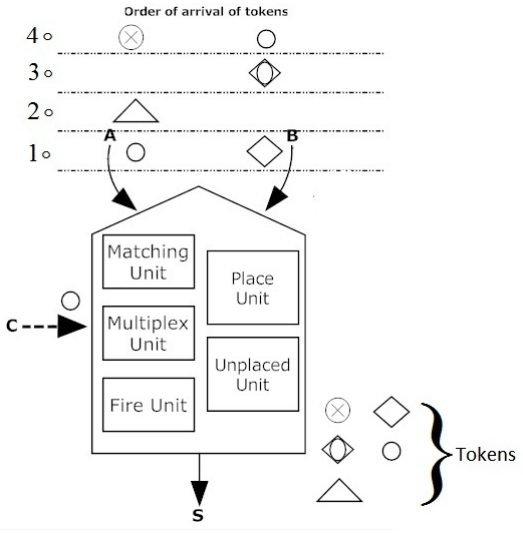

Figure 13: Tokens arrive on the branch operator

In Figure 13, a branch operator is shown, which receives various tokens in its inputs A, B and C. Creating instances depends on the tests made by the matching unit, as discussed above. Each time a token arrives, the coincidence of the tag of this token is verified the tokens stored in the matching unit of the branch operator. If there is no coincidence, a new instance is created and the non match token is stored in the register of the matching unit of the new instance in the corresponding register to the input of the token. In Figure 14, the allocation of instances relating to tokens which arrived at the branch operator is shown.

A system as a collection of concurrent hardware dataflow nodes that communicate with each other through message-passing channels was designed.

The operators/operator and operators/instances communication use handshake protocols to send and receive tokens at the arc/channels respectively.

The ChipCflow architecture explores partial dynamic reconfiguration from Xilinx FPGAs, particularly those from the Virtex classes.

Functions which are too large to fit in one FPGA are partitioned into different sections which are successively downloaded into the FPGA in accordance with a predefined schedule. Temporary placement defines the time at which it is mapped into the FPGA for computation for each module. Additionally the position of the module inside the FPGA is given. Using partial reconfigurable devices, parts of the device can be replaced while the remaining part is still active.

Concerning application, one partial reconfigurable device can be used to save space and power. As the objects arrive at the conveyor over a given period, the device can be periodically and partially reconfigured to execute the different tasks [7].

Operators in a CDFG graph, for example, shown in Figure 9 are implemented in the FPGA without partial reconfiguration. On the other hand, the instances are created and destroyed at the runtime. These instances are temporarily implemented on the device.

Empirical analysis showed that the time spent to reconfigure each instance is high. Therefore, we are studying the possibility to instantiate a number ”n” of instances at a time, instead of one by one and decompress to reconfigure the FPGA more efficiently [8].

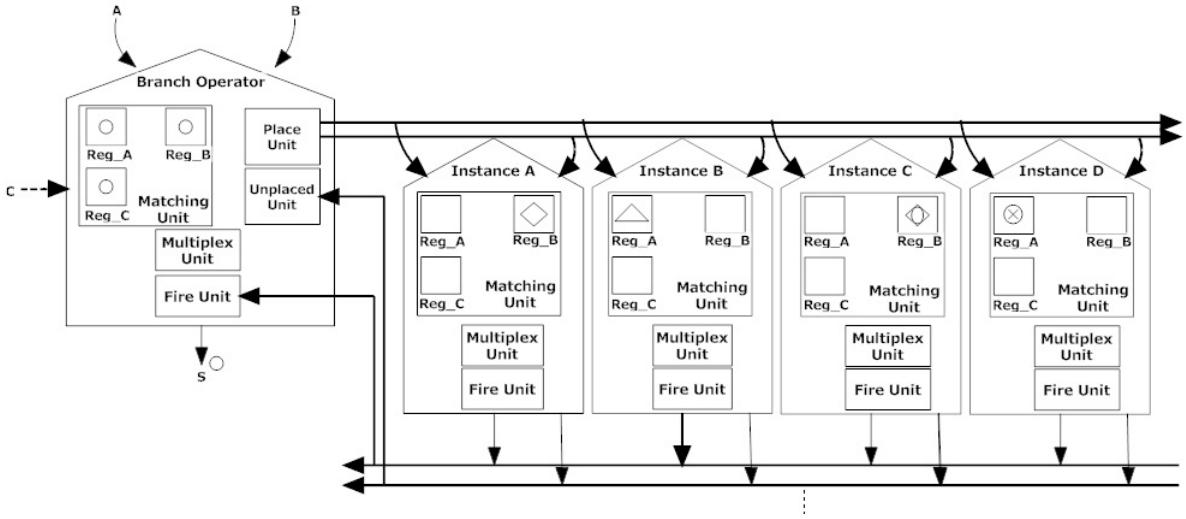

Figure 14: Operator branch with your created instances

4.3.4 Memory organization and management of data structure

For the management of data structures and control iterative constructors, Chipcflow has three operators: the NTM *(New Tag Manager)*; NIG *(New Iteration Generation)* and NTD *(New Tag Destructor)*.

A new tag is generated for the operator NTM when the data goes into a program, function or procedure generating a new activation in the tag shown in Figure 10. The NTM operator also removes the tag when the data leaves the program, function or procedure. The NIG operator modifies a tag generating a new value for the iteration. The NTD operator modifies a tag of the data turning it back to the correspondent level of activation before the input in the last iterative constructor and sends this information to the NTM correspondent to that level.

The NTM, NIG and NTD modify the fields (activation, nesting and iteration) of the token, thus controlling the movement of tokens in the operators.

If there is a loop implemented with iterative operations (”WHILE”, ”REPEAT” and ”FOR”), the input of data into the loop generates a new tag that is associated to the old tags. For each cycle of the loop, the tag must be adjusted to indicate a new iteration. As the iterative operations can be nested, the tag also has the input of data in the loop which will generate a new tag. For each cycle of the loop, the tag must be adjusted to indicate a new iteration. As the iterative operations can be nested, the tag also has the level of the nest. At the end of each iterative constructor, part of the tag is modified indicating that the tag is leaving that level of activation tag. In Figure 15, a program using the iterative construction is shown.

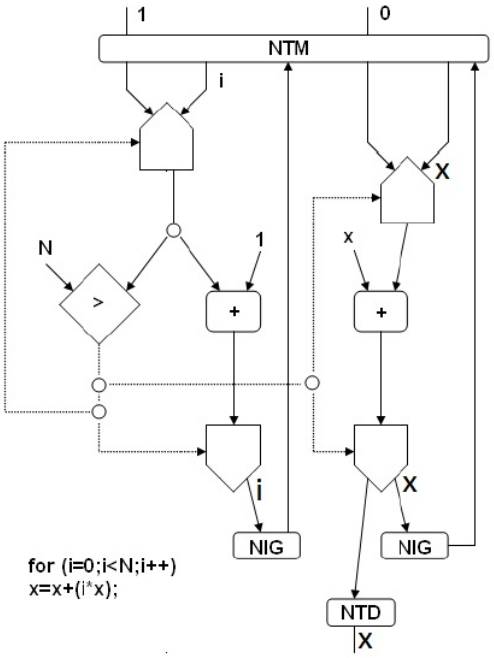

Figure 15: the Chipcflow of the program using iterative constructors

The compiler creates Chipcflow memory banks for each program variable and especially with the NTM operator, which directly accesses the program variables in the memory enabling the modification of a single element of an array or data structure, for example. Thus, is not necessary the creation of another array identical to the original to keep the elements of the array sorted.

The organization of memory depends on the compiler which analyzes the source program. There may

be the creation of memory which is centralized, distributed or simply registers for constant variables.

Although there are various organizations of memory in the various NTM distributed CDFG graph. As each loop, function or procedure has a NTM, the organization of memory depends on the complexity of these blocks.

5 Conclusion

The major contributions of this paper are the description of the ChipCflow tool, which convert C directly into the hardware in a dynamic dataflow architecture, using a dynamic FPGA reconfiguration. Initially various considerations are reflected the traditional dataflow architecture and the contemporary dataflow architecture. Therefore, the advantages of using ChipCflow, mainly in the data structure are described. The best advantage of the dataflow architecture presented in this paper is parallelism, associated with a mechanism of instances and the individual structures of matching stores for each instance, compared to the centralized matching stores present in traditional dataflow architecture. Another advantage of the chipcflow architecture is to use a specific operator for data structures as a vector and matrix. Normally, it is necessary to use two vectors or two matrices, one used as an input, and another generated with the output. In the ChipCflow, the same array is used, the difference is the specific operator for that.

References:

[1] D. Abramson and G. Egan. The rmit data flow computer: a hybrid architecture. *Comput. J.*, 33(3):230–240, 1990.

[2] Arvind and David E. Culler. Dataflow architectures. pages 225–253, 1986.

[3] Arvind and Robert A. Iannucci. A critique of multiprocessing von neumann style. *SIGARCH Comput. Archit. News*, 11(3):426–436, 1983.

[4] A. Arvind and K. P. Gostelow. The u-interpreter. *Computer*, 15(2):42–49, 1982.

[5] A. Arvind and K.P. Gostelow. The u-interpreter. *Computer*, 15(2):42–49, 1982.

[6] D. E. Iannucci R. A. Kathail V. Pingali K. Thomas R. E. Arvind, Culler. The tagged token dataflow architecture. 1983.

[7] Christophe Bobda. Synthesis of dataflow graphs for reconfigurable systems using temporal partitioning and temporal placement. Master's thesis, University of Paderborn, May 2003.

[8] Christophe Bobda. *Introduction to Reconfigurable Computing: Architectures, Algorithms, and Applications*. Springer Publishing Company, Incorporated, 2007.

[9] Andrea Cappelli, Andrea Lodi, Claudio Mucci, Mario Toma, and Fabio Campi. A dataflow control unit for c-to-configurable pipelines compilation flow. In *FCCM '04: Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 332–333, Washington, DC, USA, 2004. IEEE Computer Society.

[10] J. B. Dennis and G. R. Gao. An efficient pipelined dataflow processor architecture. In *Supercomputing '88: Proceedings of the 1988 ACM/IEEE conference on Supercomputing*, pages 368–373, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[11] Jack B. Dennis. Retrospective: a preliminary architecture for a basic data flow processor. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 2–4, New York, NY, USA, 1998. ACM.

[12] Jack B. Dennis. Retrospective: a preliminary architecture for a basic data flow processor. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 2–4, New York, NY, USA, 1998. ACM.

[13] Jack B. Dennis, G. Andrew Boughton, and Clement K.C. Leung. Building blocks for data flow prototypes. In *ISCA '80: Proceedings of the 7th annual symposium on Computer Architecture*, pages 1–8, New York, NY, USA, 1980. ACM.

[14] Jack B. Dennis, Guang R. Gao, and Kenneth W. Todd. Modeling the weather with a data flow supercomputer. *IEEE Trans. Computers*, 33(7):592–603, 1984.

[15] V. G. Grafe, G. S. Davidson, J. E. Hoch, and V. P. Holmes. The epsilon dataflow processor. In *ISCA '89: Proceedings of the 16th annual international symposium on Computer architecture*, pages 36–45, New York, NY, USA, 1989. ACM.

[16] S. Hauck. The roles of fpgas in reprogrammable systems. *Proceedings of the IEEE*, 86(4):615–638, Apr 1998.

[17] Yasuhiro Inagami and John F. Foley. The specification of a new manchester dataflow machine. In *ICS '89: Proceedings of the 3rd international conference on Supercomputing*, pages 371–380, New York, NY, USA, 1989. ACM.

[18] Krishna Kavi, Joseph Arul, and Roberto Giorgi. Execution and cache performance of the scheduled dataflow architecture. 2000.

[19] Krishna M. Kavi, A. R. Hurson, Phenil Patadia, Elizabeth Abraham, and Ponnarasu Shanmugam. Design of cache memories for multithreaded dataflow architecture. *SIGARCH Comput. Archit. News*, 23(2):253–264, 1995.

[20] B. Lee and A.R. Hurson. Dataflow architectures and multithreading. *Computer*, 27(8):27–39, Aug 1994.

[21] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective compiler support for predicated execution using the hyperblock. *SIGMICRO Newsl.*, 23(1-2):45–54, 1992.

[22] Ramadass Nagarajan, Karthikeyan Sankaralingam, Doug Burger, and Stephen W. Keckler. A design space evaluation of grid processor architectures. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 40–51, Washington, DC, USA, 2001. IEEE Computer Society.

[23] Walid Najjar and Jason Villarreal. Reconfigurable computing in the new age of parallelism. In *SAMOS '09: Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 255–262, Berlin, Heidelberg, 2009. Springer-Verlag.

[24] RAVI K. NAMBALLA. Chess: A tool for cdfg extraction and high-level synthesis of vlsi systems. Master's thesis, University of South Florida, July 2003.

[25] Gregory M. Papadopoulos and Kenneth R. Traub. Multithreading: a revisionist view of dataflow architectures. *SIGARCH Comput. Archit. News*, 19(3):342–351, 1991.

[26] David Pellerin and Scott Thibault. *Practical FPGA Programming in C*. Prentice Hall PTR, April 2005.

[27] S. Sakai, y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An architecture of a dataflow single chip processor. *SIGARCH Comput. Archit. News*, 17(3):46–53, 1989.

[28] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore. Exploiting ilp, tlp, and dlp with the polymorphous trips architecture. *SIGARCH Comput. Archit. News*, 31(2):422–433, 2003.

[29] Costa Kelton A. P. da Roda Valentin Obac Silva, Jorge Luiz e. *The C Compiler generating a Source file in VHDL for a Dynamic Dataflow Machine*. COMPUTER and SIMULATION IN MODERN SCIENCE. Published by WSEAS Press, 2009.

[30] David F. Snelling and Gregory K. Egan. A comparative study of data-flow architectures. In *PACT '94: Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques*, pages 347–350, Amsterdam, The Netherlands, The Netherlands, 1994. North-Holland Publishing Co.

[31] Vason P Srini. An architectural comparison of dataflow systems. *Computer*, 19(3):68–88, 1986.

[32] Andrew Schwerin Steven Swanson, Ken Michelson and Mark Oskin. Dataflow: The road less complex. *In the Workshop on Complexity-effective Design (WCED) held in conjunction with the 30th Annual International Symposium on Computer Architecture (ISCA)*, 2003.

[33] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. Wavescalar. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 291, Washington, DC, USA, 2003. IEEE Computer Society.

[34] Steven Swanson, Andrew Putnam, Martha Mercaldi, Ken Michelson, Andrew Petersen, Andrew Schwerin, Mark Oskin, and Susan J. Eggers. Area-performance trade-offs in tiled dataflow architectures. *SIGARCH Comput. Archit. News*, 34(2):314–326, 2006.

[35] Steven Swanson, Andrew Schwerin, Martha Mercaldi, Andrew Petersen, Andrew Putnam, Ken Michelson, Mark Oskin, and Susan J. Eggers. The wavescalar architecture. *ACM Trans. Comput. Syst.*, 25(2):4, 2007.

[36] Arthur H. Veen. Dataflow machine architecture. *ACM Comput. Surv.*, 18(4):365–396, 1986.

[37] P.G. Whiting and R.S.V. Pascoe. A history of data-flow languages. *Annals of the History of Computing, IEEE*, 16(4):38–59, Winter 1994.