Tag Management in a Reconfigurable Tagged-Token Dataflow Architecture

BRUNO DE ABREU SILVA
University of Sao Paulo
Department of Computer Systems
Av. Trabalhador Saocarlense, 400
BRAZIL
brunoas@icmc.usp.br

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

Abstract: Combining dataflow concepts with reconfigurable computing provides a great potential to exploit the application parallelism efficiently. However, to express such parallelism cannot be a trivial task. Therefore, there is a great effort to automatically translate programs originally written in procedural languages (like C and Java) into dataflow architectures which express the parallelism in a natural way. Our previous work presents a static dataflow architecture which is part of a framework to translate C programs into reconfigurable dataflow architectures. In this paper, it is discussed an implementation of tag management in a reconfigurable tagged-token dataflow architecture which was implemented on a field programmable gate array (FPGA). Although tagged-token is a traditional concept to implement dynamic dataflow machines, they have not been well explored in FPGA-based dataflow architectures. The FPGA-based dynamic dataflow architecture shows the potential for high computation rates allowing more efficient execution and presenting a more effective way to exploit parallelism for several program statements, as nested loops and function calls, when compared to static dataflow architectures.

Key–Words: Tagged-Token Dataflow Architecture, Dynamic Dataflow Model, Reconfigurable Computing.

1 Introduction

With the advent of reconfigurable computing and Field Programmable Gate Array (FPGA), researchers are trying to explore the maximum capacities of these devices, which are: flexibility, parallelism, optimization for power, security and real time applications [1, 2]. However, the development of tools to convert algorithms into hardware description to execute in 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 [3, 4].

This paper takes a step forward from our previous work [5] where accelerating algorithms was implemented converting parts of programs written in C language by using an FPGA-based static dataflow model. However, the static model cannot exploit all parallelism in an application. In the literature, the dynamic dataflow architecture was proposed to overcome this problem although the FPGA implementation of this model was not widely explored. The main goal in this paper is to describe a specific kind of dynamic dataflow model, called tagged-token, tag management concepts, operators and hardware support implemented on FPGA to better exploit the available application parallelism.

The remainder of this paper is organized as follows. Related work is described in section 2. The dataflow graph model is discussed in section 3. In section 4, the iterative operators for the dataflow architecture is presented. Section 5 presents the operators implementations and the results are presented in Section 6. Finally, Section 7 presents the conclusion.

2 Related Work

The dataflow graph model and its architecture were first researched in the 1970s and were discontinued in the 1990s [6, 7, 8, 9]. Nowadays, they are a topic of research once more, mainly due to the advance of reconfigurable technologies, particularly with the advent of FPGAs [10, 8, 2].

Recently, Michael Flynn has published a paper [11] with very important considerations about dataflow processors and it is described here: *”In parallel software achieving multicore speedup is limited by: Efficient distribution of tasks, Inter-node communications (data assembly & dispatch), Memory bandwidth, Layers of abstraction that hide critical sources of and limits to efficient parallel execution. One approach to the problem is to use heterogeneous compute elements in the form of accelerators. The ap-*

plication code is profiled and only the kernels of the application are relocated to the accelerator. Usually in relevant applications, the kernels represent a small part of the code (usually less than 10,000 lines). If the kernels of the application represent well over 90% of the application execution time then only they need to be rewritten for the accelerator, limiting the law of effort to achieve speedup. Successful graphical accelerator implementations follow this line. There is an even greater speedup opportunity with FPGA based accelerators especially when they realize dataflow compute engines to implement the kernels".

Since the dataflow model has an implicit parallelism and the FPGA is composed by parallel circuits, the dataflow model applied to an FPGA has the perfect combination to execute applications which also have parallelism in their execution [8]. However, as applications become more complex, software development is only possible using high level languages, such as C or Java [12], although only parts of a program will be executed directly into the hardware. Therefore, several tools have been developed to convert C into hardware using VHDL (Very high speed integrated circuits Hardware Description Language) [13, 14, 15].

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 is generated in VHDL. The hardware generated using these tools consists of coarse grain elements or assembly instructions for a customized processor as Picoblaze or Nios from Xilinx and Altera, respectively [16].

Our approach relies on a fine grain model implementing a dynamic dataflow architecture in VHDL. Such architecture consists of nodes of processing elements connected by arcs forming a graph.

3 The Dataflow Graph Model

In the Asynchronous Dataflow Graph project, which was developed by Teifel et al. [2], 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 it is allowed only one sender and one receiver for each channel. Since there is no clock in an asynchronous design, processes use handshake protocols to send and receive tokens via channels.

In Fig. 1, it is described 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.

In our project, we use a collection of concurrent hardware processes that communicate with each other by using a parallel bus with bits for data and bits to control the communication in a synchronous system of communication.

3.1 Dataflow Computations

A traditional dataflow model is described in the literature and is used to accelerate algorithm designs. In a dataflow graph, a node represents a processing element and an arc represents the communication between two processing elements [6, 10, 7, 8, 9]. A data bus and a control bus to execute the communication between the operators were implemented in our previous static dataflow graph model, where only one item of data can be in an arch.

In Fig. 2, an example of a basic operator and its data buses and control buses for communication are depicted. The signal data a, b and z in Fig. 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 according to a handshake protocol.

The communication protocol between operators is described in Fig. 3. As can be seen in figure, two operators (a sender and a receiver) are communicating. Both sender and receiver operators have two input data buses a and b, one output data bus z and their respective control signals *stra*, *strb*, *strz*, *acka*, *ackb* and *ackz*. Each of the 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. The sender output z is connected to the receiver input a; the output control signal *strz* from the sender is connected to the input control signal *stra* from the receiver; and the input control signal *ackz* from the sender is connected to the output control signal *acka* from the receiver. A "logic-0" in the signal *ackz* informs the sender that the receiver is ready to receive data. A "logic-1" in the signal *ackz* informs the sender that the receiver is busy. A "logic-1" in the signal *stra* informs the receiver that an item of data is ready to be sent to it from the sender. A "logic-0" in the signal *stra* informs the receiver that the sender does not have an item of data to be sent to it. To start the communication, an *enable* signal with "logic-0" is sent to the *ackz* connected to the sender (Fig. 3a). When the receiver is ready to receive data, a "logic-1" is sent to *stra* and an item of data is sent from sender to the receiver input data bus

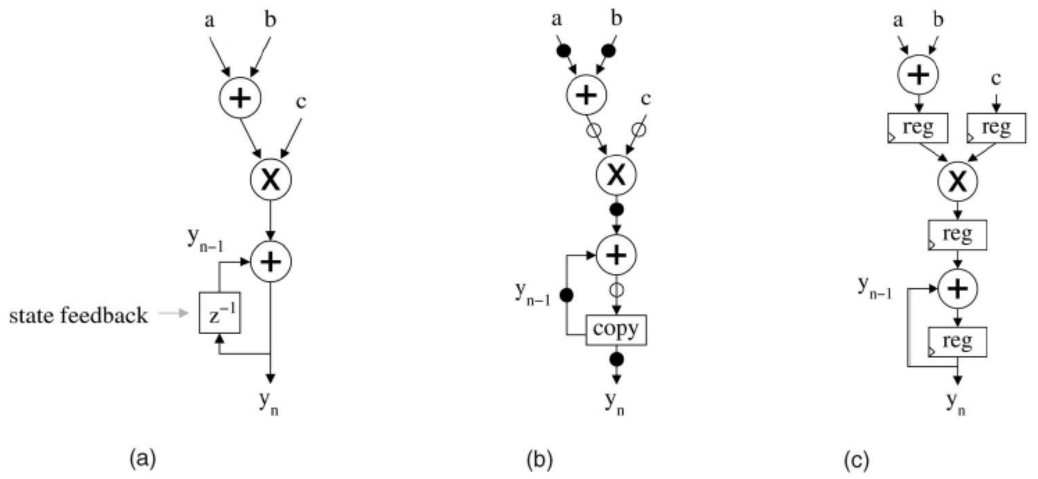


Figure 1: Computation of $y_n = y_{n-1} + c(a+b)$:(a) pure dataflow graph, (b) token-based asynchronous dataflow pipeline (filled circles indicate tokens, empty circles indicate an absence of tokens), and (c) clocked dataflow pipeline [2].

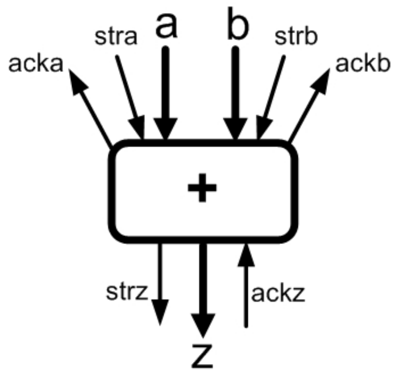


Figure 2: A basic operator with its data buses (input *a* and *b*, and output *z*) and handshake control buses (*ack* and *str* for each data bus).

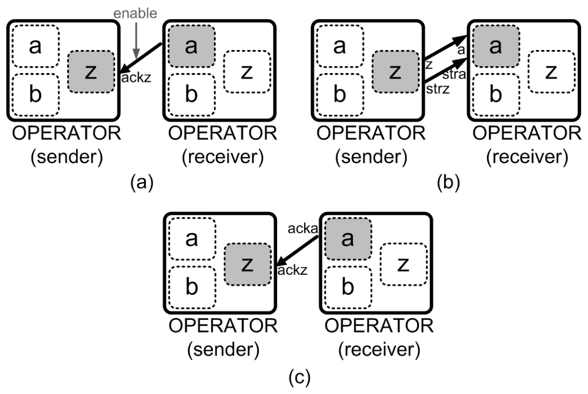


Figure 3: The communication protocol between two operators (sender and receiver): a) enabling the communication, b) sending an item of data, c) Acknowledging an item of data.

a (Fig. 3b). Finally, a "logic-0" in the *acka* acknowledges that the item of data *a* was received and stored (Fig. 3c).

3.2 The Dataflow Operators

The dataflow operators implemented in our architecture were the traditional operators described by Veen in [9], which are: copy, non deterministic merge, deterministic merge, branch, conditional and primitive operators (such as, add, sub, mul, div, and, or, not, etc.).

To perform the computation in an operator, it is necessary that all of its input buses of data presents an item of data. In Fig. 4, operators are described where filled circles indicate items of data, and empty circles show absence of items of data. Before the computation begins, data are available in all operator inputs. Then, the computation produces its results and data is sent to the operator output after the computation ends [2].

The functional execution of dataflow operators is described below:

1. *Branch*: This dataflow node performs a two-way controlled data branch and allows the item of data to be conditionally sent to two different output buses. It receives a control signal *C* which can be TRUE or FALSE used to decide which output data (*Z1* for TRUE or *Z2* for FALSE) will transfer the input data *A*;

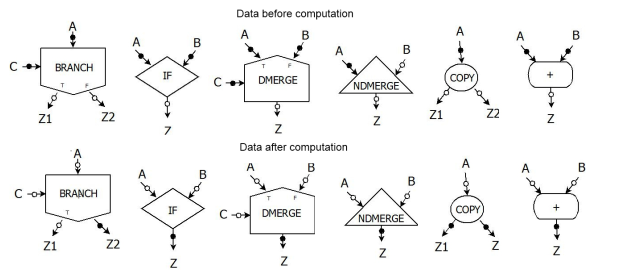


Figure 4: Basic dataflow operators (*Branch, Conditional, Dmerge, NDmerge, Copy, and Primitive*) [2]. Before the computation, all required input data (represented by filled circles) are available to the operator. Then, after the computation, the result is available in the operator output (also represented by filled circles).

2. *Conditional (If)*: Comparison operations are performed by this operator. It receives two items of data from its inputs *A* and *B* and produces a logical value TRUE or FALSE in its output *Z* according to the implemented comparison. Operations such as *equal to*, *greater than*, *less than* are implemented by using this operator;

3. *Dmerge*: This dataflow node performs a two-way controlled data merge and allows an item of data to be conditionally read from input data buses. If the control signal (*C*) is FALSE, the item of data present in input *A* is sent to the output *Z*. On the other hand, if *C* is TRUE, the item of data present in *B* is sent to the output *Z*;

4. *NDmerge*: 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 then sent to the output *Z*;

5. *Copy*: This dataflow node duplicates an item of data from the input *A* to its two outputs *Z1* and *Z*;

6. *Primitive*: This dataflow node receives two item of data in its input data buses (*A* and *B*), computes the primitive operation with these two items of data and generates the result sending it to the output data bus *Z*. Operators such as add, sub, multiply, divide, and, or, and not are implemented in the same way.

4 The Iterative Operators

There are two methods to implement a dynamic dataflow model: *code-copying* and *tagged-token*. The *code copying* method can achieve high level of parallelism by copying subgraphs of loops into the FPGA and executing them at the same time for different iterations of the loop. However this method requires a complex area and data traveling management for the subgraph. This method is similar to *loop unrolling* techniques.

The *tagged-token* method adds a *tag* to each item of data traveling by the dataflow graph. The *tag* identifies the instance where the item of data is located in the dataflow graph. Then, an operator performs computation when each input arc of that operator contains an item of data with *identical tags*.

In the Fig. 5 part a), it is described a simple iterative statement *while* where the *condition* is a function of *f(x)* and the loop command is a function of *g(x,y)*. In this case, the *f(x)* and *g(x,y)* control all the data flowing through the graph preventing deadlock. A different situation is described in the Fig. 5 part b) where the box *h* is a delayer which generates an incompatibility with data traveling into the graph. Finally, in the Fig. 5 part c), it is described a loop graph for the same algorithm in Fig. 5 b). However, the graph solves the data incompatibility by using the *tagged-token* method. As can be seen in the Fig. 5 part c), a *tagged-token* method introduces specific operators in graph to manipulate tags. A *new tag area* operator is allocated at the loop beginning to generate a tag for an item of data coming into the loop. For the iteration of the item of data, a *next tag* operator modify the tag to informs the new iteration. Finally, at the end of the loop, the *tag restore* operator restores the old data tag, that means, the tag that the data had before entering the loop [9].

4.1 Tag Management Operators

In Fig. 6, it is described a subgraph G_1 representing a *for* command. In Fig.7, it is described another *for* command with subgraph G_1 inside it. As can be seen in the Fig. 7, for each stream of the variables i and m, from an external *for* command, a new activation for the subgraph G_1 occurs. Once activated, the item of data i and the item of data m flow through the subgraph normally, and various computation iterations over the item of data i and the item of data m can occur.

The whole graph (the whole program), as well as a particular subgraph (functions, procedures, and loop statements), can have items of data belonging to different iterations and activations flowing into its structure. Therefore, three different tags are needed to ensure the right algorithm execution: a tag representing the whole graph activation (*graph_act*), the subgraph activation (*subgraph_act*), and the loop iteration (*cur-*

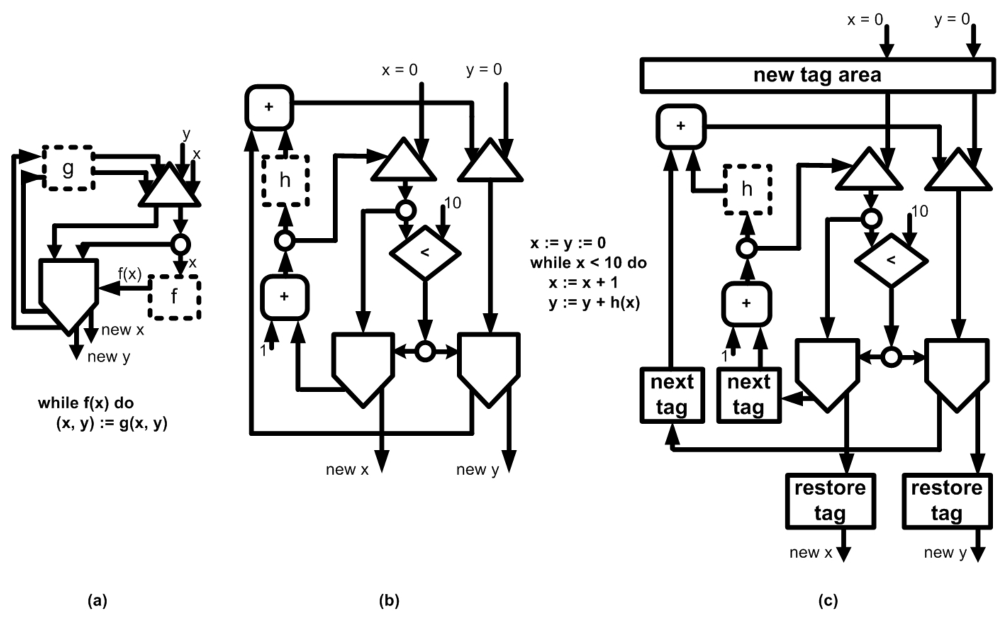


Figure 5: Different models of dataflow iterative constructs.

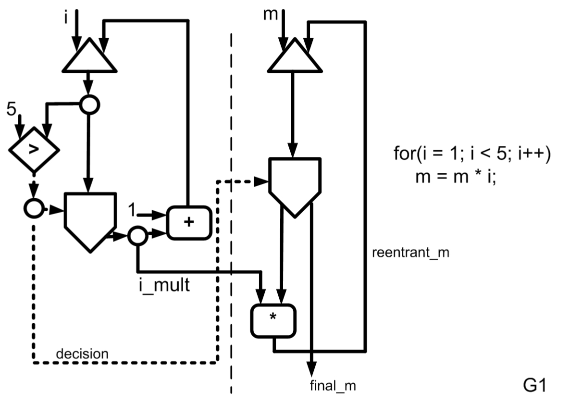


Figure 6: Example of a subgraph G_1 representing an iterative construction graph.

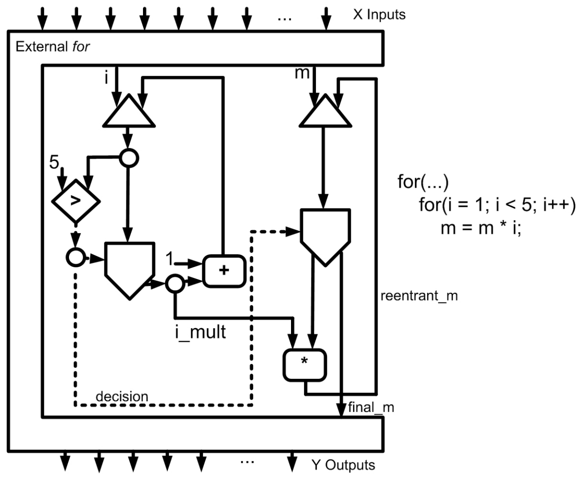


Figure 7: Example of nested loops.

rent_it). The dataflow graph operators used to manage such tags, which were implemented in this work, are presented in the following sections.

4.1.1 Initial Tag Generator - ITG

The ITG operator is responsible to generate an initial tag for each item of data coming into a graph. The ITG operator has a register (*graph_act_reg*) that stores the current graph activation. Initially, the register value is zero, and it is incremented in every new graph activation. A new activation occurs whenever new items of data arrive to ITG operator. When an item of data receives all tags, the ITG provides the *graph_act_reg* value for its *graph_act* tag. The ITG also initializes the other tags. Therefore, *subgraph_act* and *current_it* receive zero as initial value. As the ITG generates initial tags for data, it is an operator extremely important to ensure that each item of data will be processed only with its partners.

In Fig. 8, it is described an example using an ITG operator. As can be seen in figure, the graph is supplied with data to be processed for three variables: A, B, and C. The arriving order is important since it defines the tag value. Therefore, a_0, b_0, and c_0 must arrive before a_1, b_1, and c_1 respectively. Our compiler generates the data in the right order. As soon as the a_0 arrives, its value is concatenated to the respective tag with the values (0,0,0) representing *graph_act*, *subgraph_act* and *current_it* respectively. When a_1 arrives, the tag with the values (1,0,0) is concatenated to

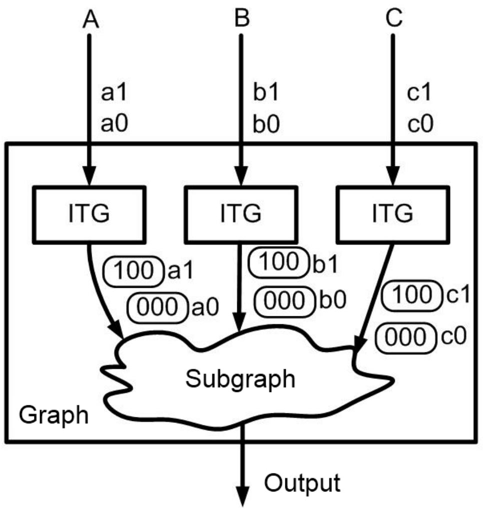


Figure 8: The initial tag generated by ITG operator.

a_1 since a_1 belongs to a new activation for the subgraph. The tag generation for the other data is similar. Each variable being processed into the graph has its own ITG. In this way, only the arriving order is important, however there is no need to all variables arrive into the graph at the same time.

4.1.2 Tag Remover - TR

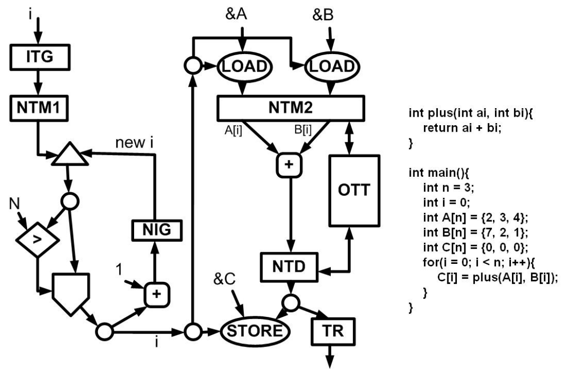
Whenever an item of data leaves the graph, it does not need tag anymore. Therefore, the tag can be removed. The *Tag Remover* (TR) operator removes the tag of an item of data.

4.1.3 New Iteration Generator - NIG

After an item of data coming into a graph, it can be inside a loop, and the *current_it* tag need to be adjusted for each iteration of the data. Then, the *New Iteration Generator* (NIG) adjusts the *current_it* for each iteration of the item of data, incrementing this tag.

4.1.4 New Tag Manager - NTM

The *New Tag Manager* (NTM) assigns a new subgraph activation to the tag of the items of data coming into a subgraph (a subgraph can be a function body or loop statements). The NTM has a register (*subgraph_act_reg*) that starts with zero and for each new subgraph activation its value is incremented. Whenever an item of data pass by NTM, the value of register *subgraph_act_reg* is assigned to the *subgraph_act* tag of data. When the item of data finished to be processed in the subgraph, it needs to recover the tag it



```
int plus(int ai, int bi){
   return ai + bi;
}

int main(){
   int n = 3;
   int i = 0;
   int A[n] = {2, 3, 4};
   int B[n] = {7, 2, 1};
   int C[n] = {0, 0, 0};
   for(i = 0; i < n; i++){
      C[i] = plus(A[i], B[i]);
   }
}
```

Figure 9: Sum of vectors element by element.

had before entering the subgraph in order to continue to be rightly processed in other graph parts. Therefore, the *old tag* must be stored in some way. The next section explains the process to recover old tags.

Differently from what happens in ITG, it is not always possible to ensure the right arriving order for data partners in NTM. Therefore, instead of having a particular NTM for each variable, we have only one NTM for all variables used in a given subgraph. Then, the NTM synchronizes the items of data coming into the subgraph with the same *subgraph_act* tag (the partners). When all partners are available, the NTM supplies its subgraph with such data. As new loop iterations can occur in this subgraph, NTM operator assigns zero to the *current_it* tag.

4.1.5 New Tag Destructor - NTD

Whenever an item of data pass by a NTM, the *current_it* tag is changed, and the old tag before the item of data coming into the subgraph is lost. In order to recover the old tags, a memory is used to store the tags before to pass by the NTM operator. This memory is called *Old Tag Table* (OTT). OTT stores two information for each input set: old tag which is composed by subgraph activation (*subgraph_act*) and current iteration (*current_it*); and new tag (represented by graph activation and new subgraph activation - assigned by NTM). The NTM saves the old tag in OTT to be recovered later. NTD consults OTT to recover the old tag, whenever it receives an item of data.

5 Implementation and Synthesis

In Fig. 9, it is described a dataflow graph that uses all tag management operators proposed in this project. The C code related to the dataflow graph has a function called *plus* that receives two parameters and returns the sum of them. The main program function has a *loop* that calls the function plus for each element of vectors A and B and stores the results in vector C.

At the beginning of graph execution, the constants (address of vectors A, B, C and values of variables i and n) are loaded into memory. All constants are waiting for their partners while i enters ITG and receives its initial tag $< 0.0.0, 0 >$ representing *graph_act*, *subgraph_act*, *current_it* and the i value.

When the variable i enters NTM1, its subgraph activation tag receives the value of NTM1 subgraph activation, that is zero. The NTM1 refers to the *loop* beginning in C code. If the logic comparison inside the loop returns TRUE, the sum operator increments i, the NIG increments current iteration of tag and the token becomes $< 0.0.1, 1 >$. At the same time, a copy of i arrives in *load* and *store* operators. The *store* then remains waiting for the item of datum which is partner of i. After *load* execution, $A[0]$ and $B[0]$ are sent to NTM2, that refers to function plus beginning.

NTM2 receives $A[0]$ and $B[0]$ and the values of tokens are $< 0.0.0, 2 >$ for $A[0]$ and $< 0.0.0, 7 >$ for $B[0]$. Their old tags are associated to the new ones and are stored in OTT. When the sum result of $A[0]$ and $B[0]$ arrives in NTD, the old tag is recovered, the store instruction receives the missing partner of i copy and store the item of data in $C[0]$. Finally, a copy of the result is sent to TR that removes the tag and the data can be sent to some output device, for instance. The execution cycle continues until the logic comparison between i and n returns FALSE.

Listing 1: VHDL piece of code illustrating an ITG *case* statement with two states: sending and receiving data.

```
case state is
  when receiving_data =>
    if input_str = '1' then
      input_reg := input;
      state := sending_data;
    end if;
  when sending_data =>
    if output_ack = '1' then
      graph_act_reg := graph_act_reg + 1;
      state := receiving_data;
    else
      output(graph_act_size - 1 downto 0)
          <= graph_act_reg;
      output(tag_size - 1 downto
          graph_act_size) <= (OTHERS =>
          '0');
      output(token_size - 1 downto
          token_size - data_size) <=
          input_reg;
      output_str <= '1';
      input_ack <= '1';
    end if;
end case;
```

The operators were implemented by using a finite state machine described in VHDL. A piece of code of ITG can be seen in Listing 1 as an example of two typical states of an operator: sending and receiving data. All operators have an initial state to receive data. When all the operands are received, the operator execution starts. At the end of execution, there is a state to send the result to output signals. In this particular implementation, a token has 32 bits: 16 for value of data and 16 for tag. The tags graph activation and subgraph activation have 4 bits and the current iteration has 8 bits. Such values for the tag length are arbitrarily chosen only for tests and to verify the operators behavior.

All operators have an internal clock signal, however the communication between operators is asynchronous according to the handshake protocol. In other words, for each output signal s_i of an operator, there is a signal, called $strobe_i$, that indicates to next operator that exists an item of data in output s_i ready to be consumed. The operators also have acknowledge signals related to input. Such signals (*ack*) indicate to previous operator that a given item of data was received.

6 Results

The tag manipulators (ITG, TR, NTM, NTD, NIG and OTT) were described in VHDL, synthesized and validated using the Xilinx ISE 9.1. The Spartan 3E FPGA family was used and the 3s500efg320-4 device was selected. Table 1 presents synthesis results of each tag manipulator.

In previous work [5], where accelerating algorithms were proposed, the Fibonacci sequence was implemented by using two others tools, C-to-Verilog and LALP, to be compared with the accelerating algorithms. In this paper, the comparison was realized to the same tools just for Fibonacci sequence and iterative operators. The Fibonacci algorithm is described in Algorithm 1 and its dataflow graph is described in Fig. 10.

Algorithm 1 Calculate Fibonacci

```
$first \Leftarrow 0$
$second \Leftarrow 1$
$tmp \Leftarrow 0$
for $i = 0$ to $n$ do
    $tmp \Leftarrow first + second$
    $first \Leftarrow second$
    $second \Leftarrow tmp$
end for
```

Table 1: Synthesis results for the VHDL implementation of each tag operators using Xilinx Spartan 3E 3s500efg320-4

	Device Elements Utilization					
	NTM	**NTD**	**ITG**	**TR**	**NIG**	**OTT**
# Slices	159 (3%)	47 (1%)	45 (0%)	21 (0%)	46 (0%)	13 (0%)
# Slices Flip-Flops (FFs)	243 (2%)	81 (0%)	74 (0%)	35 (0%)	69 (0%)	18 (0%)
# 4 input Look-Up Tables (LUTs)	122 (1%)	57 (0%)	10 (0%)	4 (0%)	41 (0%)	10 (0%)
# bounded Input-Output Blocks (IOBs)	229 (98%)	93 (40%)	53 (22%)	37 (15%)	69 (29%)	49 (21%)
# Memory Blocks (BRAMs)	-	-	-	-	-	1 (5%)
# Global Clocks (GCLKs)	1 (4%)	1 (4%)	1 (4%)	1 (4%)	1 (4%)	1 (4%)
Min period (ns)	4.786	3.523	3.126	3.056	4.390	3.561
Max Freq (MHz)	208.923	283.889	319.944	327.241	227.788	280.824

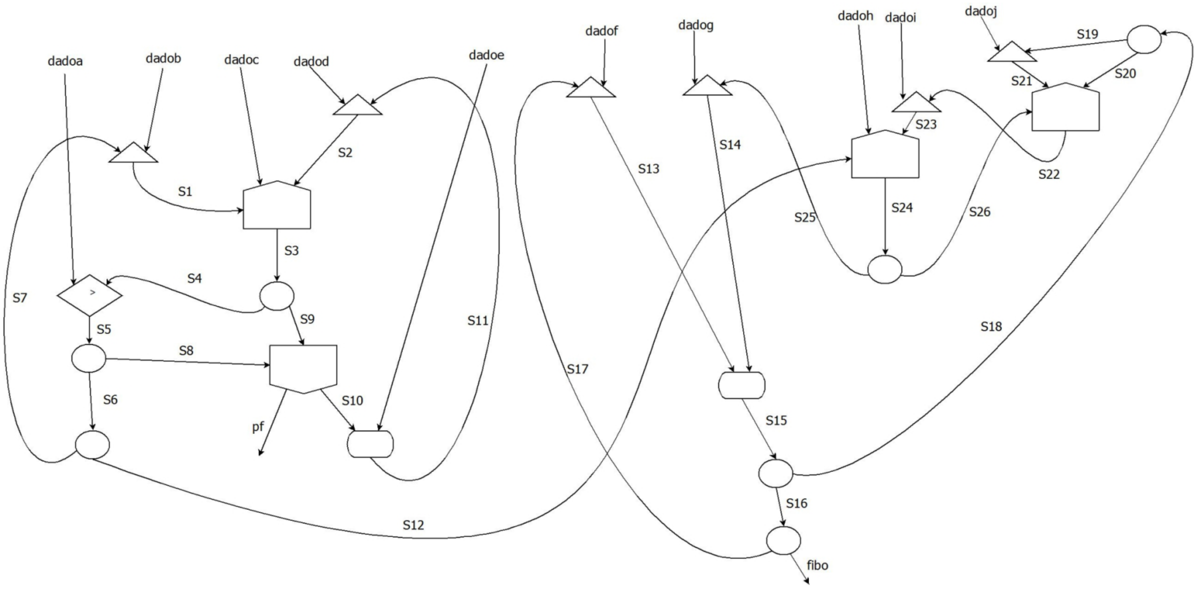


Figure 10: The Fibonacci algorithm described in Dataflow Graph.

As can be seen in Fig. 10, there are two parts in the dataflow graph: one of them is located on the left side of figure and controls the loop with index *i*; on the right side of figure, the manipulation of other variables and the calculus of Fibonacci sequence itself are described.

As the dataflow graph consists of nodes and arcs, each node represents an operator and each arc represents the communication between two operators. In Fig. 10, a label is assigned to each arc in the dataflow graph. As arcs represent the communication between two operators, the parallel data bus for items of data and the control data bus for control the communications are included in the label representations. The assembly language (mnemonic representation) that uses the name of the operator and its label arcs to convert the dataflow graph into VHDL was developed. The assembly language for Fibonacci dataflow graph is described in Listing 2. This specific assembly-like language is an intermediate dataflow representation that simplifies the compilation process for both front-end and back-end. Specifically, the compiler back-end just needs to directly translate the assembly code into VHDL. In addition, the assembly language is interesting since it can be used to automatically generate a graphical representation to debugging, mainly for large dataflow graphs.

As can be seen in Listing 2, several node operators and their input and output arcs are listed. Labels used to connect nodes operators are described initializing with the *s* character followed by a number and the others are input or output data signals. The same organization is used for the others benchmarks implementation.

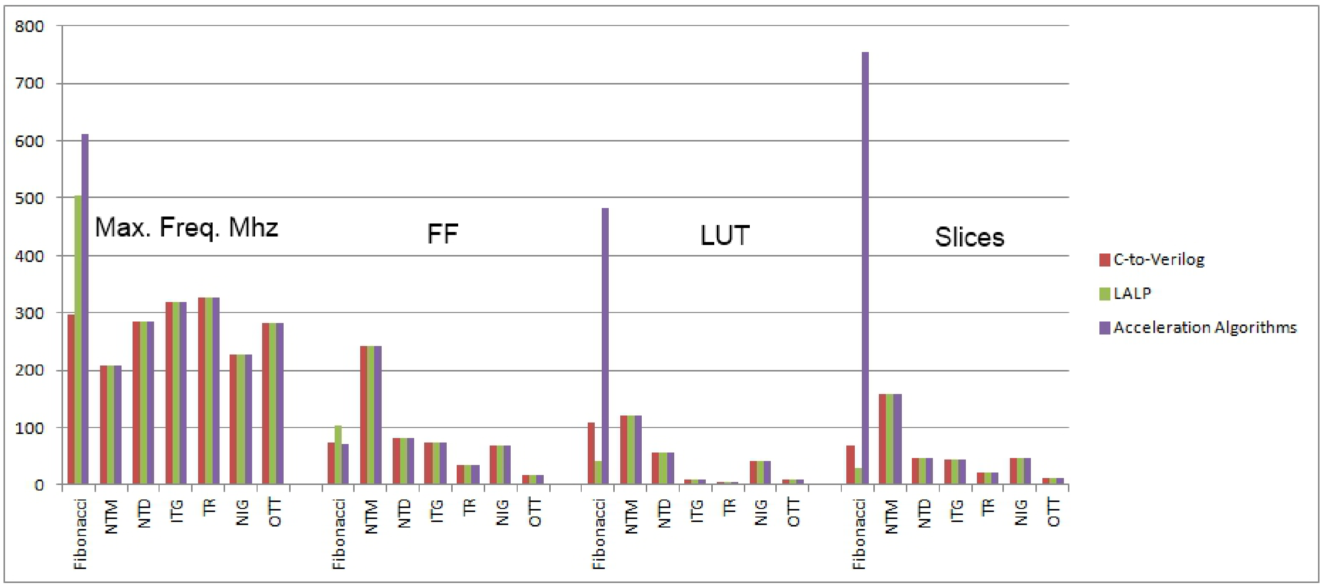


Figure 11: Comparison among the resource usage for Fibonacci sequence, Iterative Operators, C-to-Verilog, LALP, and Acceleration Algorithms according to maximum frequency achieved (Max. Freq. MHz), number of flip-flops (FF), number of Look-Up Tables (LUT), and number of slices (Slices). The y-axis represents the absolute resource usage in each case, except for Max. Freq where it represents MHz.

Listing 2: *The Assembly Language for Fibonacci Dataflow Graph*

```
1.ndmerge s7,dadob,s1;
2.dmerge s2,dadoc,s1,s3;
3.ndmerge dadod,s11,s2;
4.gtdecider dadoa,s4,s5;
5.copy s3,s4,s9;
6.copy s5,s6,s8;
7.branch s9,s8,s10,pf;
8.copy s6,s7,s12;
9.add s10,dadoe,s11;
10.ndmerge s17,dadof,s13;
11.ndmerge dadog,s25,s14;
12.ndmerge dadoi,s22,s23;
13.ndmerge dadoj,s19,s21;
14.copy s18,s19,s20;
15.dmerge s23,dadoh,s12,s24;
16.dmerge s20,s21,s26,s22;
17.copy s24,s25,s26;
18.add s13,s14,s15;
19.copy s15,s16,s18;
20.copy s16,s17,fibo;
```

In the Listing 2 the labels *dadoa*, *dadob*, *dadoc*, *dadod*, *dadoe*, *dadof*, *dadog*, *dadoh*, *dadoi* and *dadoj* are input data signals used to initialize data for the Fibonacci dataflow graph and the label *fibo* is output data signal to inform the result of the Fibonacci sequence. Specifically for the Fibonacci sequence, *dadoa* receives and maintain the *n* Fibonacci argument; *dadob* and *dadoc* receive ”logic-0” to initialize the i value in the *for* command; *dadod* receives ”logic-0” and *dadoe* receives and maintain ”logic-1” to control the next value for ”i”; *dadof* receives ”logic-1” and *dadog*,*dadoh*,*dadoi* and *dadoj* receive ”logic-0” to initialize the Fibonacci algorithm.

In Fig. 11, it is described the comparison results. As can be seen, in a general way, the iterative operators occupy less FPGA resources when compared to the other tools. However, the iterative operators have less speed than the complete benchmark Fibonacci implemented using three different tools.

7 Conclusion and Future Work

A dynamic dataflow model was implemented by using tagged-tokens to identify items of data in different positions in the dataflow graph. Besides considering the traditional operators of the dataflow model, iterative operators were implemented supporting the dynamic structure proposed in this project. The iterative operators were compared with three different tools executing a Fibonacci sequence and occupy less FPGA resources than the other tools. This is particular important in FPGA-based dataflow architectures since they are usually composed by several instances of their operators. However, the iterative operators have less speed than the complete benchmark Fibonacci implemented using three different tools. To get high performance in the dynamic dataflow model, using the iterative operators, a strong effort is necessary to reduce the speed of the iterative operators to justify a dynamic dataflow model. It is also possible to justify the dynamic model when applying it in highly parallel applications different from Fibonacci sequence. Nevertheless, the main aim in this work was to validate the implementation model. Taking this into account, dynamic dataflow model becomes a relevant solution for parallelism in FPGA.

Future work includes the development of a module to convert C into a VHDL code directly, associated with the FPGA, to implement a complete dynamic dataflow model integrating traditional operators with iterative operators in a unique tool and to compare the dynamic model to the static model when executing a set of extremely parallel benchmarks.

References:

[1] Hauck, S. (1998) The Roles of FPGAs in Reprogrammable Systems, *Proceedings of the IEEE*, IEEE, vol. 86, pp. 615–638.

[2] Teifel, J. Manohar, R. (2004) An asynchronous dataflow FPGA architecture, *IEEE Transactions on Computers*, IEEE, vol. 53, no. 11, pp. 1376–1392.

[3] 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*, Monterey, California, USA, ACM, pp. 33–36.

[4] 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 GPUs , *18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, ACM, USA, pp. 11–18.

[5] Silva, J. and Silva, B. A. and Lopes, J. J. and Silva, A. C. F. (2012) The ChipCflow Project to Accelerate Algorithms using a Dataflow Graph in a Recongurable System. *WSEAS Transaction on Computer*, WSEAS, vol. 11, pp. 265–274.

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

[7] 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.

[8] 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, vol. 25, no. 2, pp. 4:1–4:54.

[9] Veen, A. H. (1986) Dataflow Machine Architecture, *ACM Computing Surveys*, ACM, vol. 18, no. 4, pp. 365–396.

[10] 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.

[11] Michael J Flynn, Oliver Pell and Oskar Mencer (2012) DATAFLOW SUPERCOMPUTING, *22th International Conference on Field Programmable Logic and Applications FPL*, IEEE, USA, pp.1–3.

[12] Cardoso, J., H. Neto (2003) Compilation for FPGA Based Reconfigurable Hardware, *IEEE Design Test of Computers*, IEEE, vol.20, no.2, pp. 65–75.

[13] ImpulseC (2015) *Impulse Accelerated Technologies*, Impulse Accelerated Technologies, Inc. Available at: http://www.impulseaccelerated.com/. Acessed in: 04/24/2015.

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

[15] Suif (2015) *The Stanford SUIF Compiler Group*, Suifcompiler system. Available at: http://suif.stanford.edu/. Acessed in: 04/24/2015.

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