System Level Synthesis Of Dataflow Programs: HEVC Decoder Case Study

Mariem Abid[†*], Khaled Jerbi[*], Mickaël Raulet[*], Olivier Déforges[*], Mohamed Abid[†]
[*]IETR, INSA Rennes, CNRS UMR 6164, UEB
20, Av. des Buttes de Coësmes, 35708 Rennes
{mabid, kjerbi, odeforge, mraulet}@insa-rennes.fr
[†]CES Lab., National Engineering school of Sfax
Route Soukra km 4, 3000 Sfax Tunisia
{mohamed.abid}@enis.rnu.tn

***Abstract*—While dealing with increasing complexity of signal processing algorithms, the primary motivation for the development of High-Level Synthesis (HLS) tools for the automatic generation of Register Transfer Level (RTL) description from high-level description language is the reduction of time-to-market. However, most existing HLS tools operate at the component level, thus the entire system is not taken into consideration.**
We provide an original technique that raises the level of abstraction to the system level in order to obtain RTL description from a dataflow description. First, we design image processing algorithms using an actor oriented language under the Reconfigurable Video Coding (RVC) standard. Once the design is achieved, we use a dataflow compilation infrastructure called Open RVC-CAL Compiler (Orcc) to generate a C-based code. Afterward, a Xilinx HLS tool called Vivado is used for an automatic generation of synthesizable hardware implementation.
In this paper, we show that a simulated hardware code generation of High Efficiency Video Coding (HEVC) under the RVC specifications is rapidly obtained with promising preliminary results.

***Keywords*—System level, dataflow, RVC, HLS, HEVC**

I. Introduction

Nowadays, the implementation of complex processing applications including image and video coding applications on heterogeneous embedded systems, becomes a more and more challenging subject. Video coding is the process of compressing and decompressing a digital video signal. The main goal of digital compression is to reduce the data size for storage, processing or transmission. With the purpose to ensure that compliant encoders and decoders can successfully inter-work with each other, it has been necessary to define a number of key international standards for image and video compression. Thereby, there exist several notable compression standards including MPEG-1, MPEG-2, MPEG-4, H.263, H.264-AVC, High Efficiency Video Coding (HEVC), etc. Knowing that the past structure of video compression standards is based on monolithic programming (usually in the form of C/C++ programs) which has many shortcomings and unsuited for hardware design, efforts have focused on standardizing a library of video coding components called Reconfigurable Video Coding (RVC) [1]. The RVC framework is based on the usage of a new actor/dataflow oriented language called CAL Actor Language (CAL) [2] that presents several advantages versus the classical imperative sequential languages.
In this context, several issues could be raised namely why dataflow programs result in a more efficient hardware implementation compared with the manual Hardware Description Language (HDL) code, and how to translate the dataflow programs into Register Transfer Level (RTL) descriptions suitable for implementation in programmable hardware. Several works have sought to address these issues, for the sake of reducing the complexity and time-to-market, and obtaining performance/cost efficient implementation. The goal is, therefore, to design video codec on higher abstraction level under the RVC framework and use High-Level Synthesis (HLS) [3] tools to automatically generate RTL code that targets different platforms (e.g., general purposes PCs, embedded systems, Field-Programmable Gate Array (FPGA), etc.).
The main contributions of this paper are in one hand a new system-level approach for generating hardware description from dataflow programs and on the other hand a new C backend for the CAL synthesizable by the newly Xilinx Vivado HLS tool that is compliant with the design flow.
The reminder of this paper is organized as follows: Section II outlines some basic properties of the dataflow programming, the Moving Picture Expert Group (MPEG)-RVC standard framework and its reference programming language, and summarizes the existing HDL code generation approaches from dataflow representations. Section III examines our proposed methodology. Section IV gives insights on experimental results by means of two different design cases. Finally, Section V presents some final conclusions and directions for future work.

II. Background and related work

In this section, we provide a brief review of the main concepts required to establish the foundation needed to understand the work presented in this paper.

A. Dataflow Programming Paradigm

In contrast to the traditional sequential programming paradigm, dataflow approach is a programming paradigm that models a program as a directed graph in which the nodes correspond to computational units and the edges represent the

This work is done as part of 4EVER, a French national project with support from Europe (FEDER), French Ministry of Industry, French Regions of Brittany, Ile-de-France and Provence-Alpes-Cote-d'Azur, Competitivity clusters Images & Reseaux (Brittany), Cap Digital (Ile-de-France) and Solutions Communicantes Securisëes (Provence-Alpes-Cote-d'Azur).

direction of the data flowing among nodes. On one hand, each computational unit's functional behavior is autonomous and independent of other computational units, thus achieving modularity, reusability and reconfigurability and easing parallelism exploitation. In the other hand, the communication semantics of the data channels and processing behavior of Functional Units (FUs) are defined by a Model of Computations (MoC) such as Kahn Process Network (KPN) [4], Synchronous Dataflow (SDF) [5] and Dataflow Process Network (DPN) [6]. We briefly review DPN as it forms the basis of the MoC used by the CAL, which is a super-set of the RVC-CAL standardized as part of the RVC framework. Derived from KPN, DPN models the data channels of the dataflow network as unidirectional unbounded First In, First Out data queues (FIFO) that have non-blocking read semantics, and the nodes as actors with a dynamic behavior. An actor is described as a set of firing rules that indicate under which conditions an actor may fire.

B. The RVC framework

Based on the dataflow programming paradigm, the MPEG standardized the RVC framework. MPEG RVC is a new specification formalism for describing video codecs in a way that it promotes flexibility and reuse. The principle of MPEG RVC consists of describing the functional and Input/output (I/O) behavior of FUs in RVC-CAL. Connections between FUs are then described by an eXtensible Markup Language (XML) Dataflow Format (XDF) to form FUs network. The FUs and the FUs network are instantiated to form an abstract model. RVC standard is specified in two parts namely MPEG-B part 4 that specifies the dataflow framework, and the MPEG-C part 4 that defines a Video Tool Library (VTL). In order to support the RVC dataflow framework, MPEG-B part 4 specifies dataflow programming language called RVC-CAL for describing FUs in a target agnostic way. In RVC-CAL, with respect to DPN semantics, FUs are implemented as actors. Each actor contains states variables, atomic actions, parameters and input and output ports that consume and produce tokens, respectively. Listing 1 gives an overview of the structure of an actor written in RVC-CAL as described below.

Listing 1: The general structure of an actor written in RVC-CAL.

```
actor FUName(<DataType> FUParam1, <DataType> FUParam2...)
<DataType> InputPort1, <DataType> InputPort2, ... ==>
<DataType> OutputPort1, <DataType> OutputPort2, ... :
// State variables.
<DataType> StateVar1;
<DataType> StateVar2;
...
// Actions.
Action1:
action InputPort1:[a], ... ==> OutputPort1:[a], ...
guard <conditions>
do
<Statements>
...
end
Action2:
action InputPort2:[a], ... ==> OutputPort2:[a], ...
guard <conditions>
do
<Statements>
...
end
// FSM schedule to control the action-selection.
schedule fsm initialState:
initialState (Action1) --> state2;
state2 (Action2) --> state3;
...
end
// Priority block to prioritize actions.
Priority
{
Action1 > Action2 > ...;
}
end
```

When an actor fires, an action is selected according to input token availability (i.e. the number of tokens necessary for its execution), *guard* conditions (as additional conditions on token values) or state variables, Finite-State Machine (FSM) and priorities. This action firing may change the state of the actor. Moreover, each actor interacts with each other by reading and writing tokens from and to FIFO as illustrated in Figure 1.

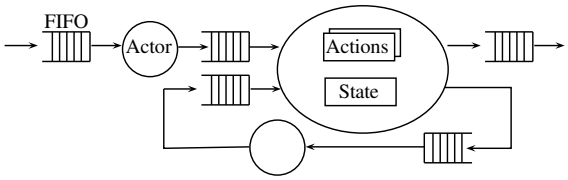

Fig. 1: A CAL network.

C. Hardware code generation from dataflow programs

The design complexity and long verification processes create a bottleneck for image and video processing applications. In order to decrease the time-to-market, many solutions were developed by raising the level of abstraction to the Electronic System Level (ESL) [7]. In Figure 2 the most common abstraction levels that are used for digital integrated circuits can be seen.

Fig. 2: The different abstraction levels.

The highest abstraction level is the system level, at which design is done taking the entire system into consideration as well, not just individual components. In other words, the interactions amongst components at an increased abstraction level are examined. Next, at the component level an algorithmic description in a high-level language is synthesized down to an RTL description. This step is commonly referred to as HLS. In the following, we examine different code generation tools available for hardware implementation whether at the component level or the system level.

1) HLS tools: At the component level, there exist several tools to perform the HLS automatically. Generally speaking, C is one possible high-level language. Here, HLS takes as an input a model written in C, C++, or SystemC, and as an output it generates a corresponding RTL representation in a HDL such as VHSIC Hardware Description Language (VHDL) or Verilog. In this case, we discuss C-to-Gate HLS tools such as Catapult C, C2H, Synphony, GAUT, etc. However, the goal of translating real-word applications written in a language such

as C into efficient hardware implementations, provides strong limitations since it doesn't take into consideration the whole system, thus it is not always efficient on an entire multi-component system description.

2) System level design: There exist also several solutions that tried to generate hardware at the system level like those starting from dataflow programs. The question arises to why dataflow programs are chosen. The efficiency of hardware generation from dataflow language (CAL) for specifying signal processing systems such as MPEG compression technology and the advantages of such approach versus the state of the art approaches were established in [8], [9]. Currently, two tools support the code generation of RVC applications towards HDL as summarized in Figure 3.

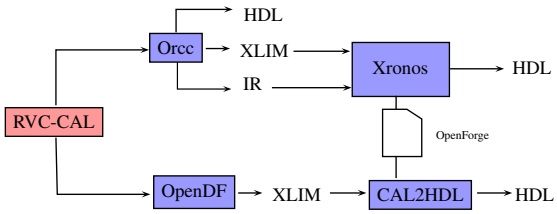

Fig. 3: RVC-CAL development tools for hardware generation.

The Open RVC-CAL Compiler (Orcc) Orcc [10] is an open-source framework to compiling network of actors and generating code for software targets (C, C++, Java, etc) as well as hardware targets (VHDL) [11]. In the front-end of Orcc, a graph network and its associated CAL actors are parsed into an Abstract Syntax Tree (AST) and then transformed into an Intermediate Representation (IR) that undergoes typing, semantic checks and several transformations in the middle-end and in the back-end. Pretty printing is applied on end on the resulting IR to generate a chosen implementation language. The limitation of the VHDL code generation is the fact that it is not mature enough to handle with all RVC-CAL structures and features.

The Open DataFlow Environment (OpenDF) OpenDF [12] is a compilation framework, its particularity is to incorporate a generator of IR in XML Language-Independent Model (XLIM) from dataflow programs (i.e. CAL to XLIM). OpenDF includes also a back-end for transforming XLIM representation to Verilog and HDL representation for hardware platforms (OpenForge) [8]. Often in RVC-CAL literature this tool is known as CAL2HDL. However, XLIM OpenDF code generation does not support all the RVC-CAL subsets, thus the choice of creating an Orcc XLIM back-end [13], [14]. Another alternative is under development, which is the Xronos [1] tool that takes as input an Orcc IR. Indeed, Xronos is an improved version of OpenForge that take into consideration the FIFO management.

For the purposes of this paper, system level design is going to refer primarily to the employed level of abstraction.

III. Proposed Method

This paper provides two main contributions concerning the automatic generation of hardware implementations from RVC-CAL descriptions. The first one deals with a C back-end of Orcc synthesizable by Vivado HLS tool, qualified "c-HLS" in the rest of this paper. The second main contribution consists in defining the system level design. Vivado HLS [2] is a Xilinx tool that transforms a C based description into a corresponding RTL representation in a HDL such as VHDL or Verilog. It turns out that Vivado HLS has a significant advantage since it speed up productivity for the 7 series devices and many generations of FPGAs to come. In the following, we detail the code generation specifications of c-HLS while respecting the DPN MoC, in order to keep the same actor behavior in the hardware description. Then, we provide the methodology used in order to establish the system level by accurately connecting the hardware generated components by Vivado HLS with the corresponding hardware FIFO components.

A. c-HLS back-end

The idea is to combine Orcc and Vivado HLS tool to obtain a full conception flow from RVC-CAL to HDL as summarized in Figure 4.

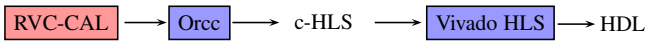

Fig. 4: Integration of Vivado HLS in the design flow.

The first characteristic of the c-HLS back-end of Orcc is the fact that it does not contain constructs which are not synthesizable such as dynamic memory allocation and pointers since RVC-CAL descriptions don't already support such constructs. Moreover, the c-HLS back-end respects the DPN MoC in that it provides FIFO management explained in the following.

1) Actor firing translation: In order to model the FIFO structure in the c-HLS back-end, we use a C++ template class, hls::stream<>, provided by Vivado HLS. However, the stream is a typical FIFO that returns only two information about its state: full or empty. Since the information about the number of tokens in the FIFO and the value of these tokens is not accessible, which proves crucial for actor firing rules as mentioned in section II-B, the solution was to create an internal circular buffer for each input port where tokens can be pulled and checked. For this issue, a similar transformation of RVC-CAL actors has already been developed in a previous work [15] and detailed subsequently based on a simple example. The Listing 2 is an "add" actor which consumes one token from its input port A and one token from its input port B unless its token value equals zero, computes the sum and produces the result in port C.

Listing 2: add actor example.

```
actor addition() int A, int B ==> int C:
        action A:[a] , B:[b] ==> C:[a+b]
                guard B = 0
        end
end
```

Formally, an actor is defined with a pair $< f, R >$ such as:

- $f : \mathbb{S}^m \rightarrow \mathbb{S}^n$ where f is a firing function that consumes sequences of tokens on m input ports and

[1] https://github.com/orcc/xronos

[2] http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/index.htm

produces sequences of tokens on n output ports and $\mathbb{S}$ is the set of all possible sequences,

- The actor can have N firing rules $R = [R_1, .., R_N]$,
- The actor can fire if and only if one or more of the firing rules is satisfied, where each firing rule constitutes a set of patterns, one for each of m inputs, $R_i = \{R_{i,1}, .., R_{i,m}\}$,
- The symbol "*" will denote a token wildcard.

Th "add" actor has only one firing rule, $R_1 = \{R_{1,1}, R_{1,2}\}$ where,

$$\begin{cases} R_{1,1} = [*] \\ R_{1,2} = [* = 0, *] \end{cases} \quad (1)$$

meaning that each of the two inputs A and B must have at least one token, with a further condition on B. Indeed, the transformation creates for each input port an internal circular buffer A_buffer and B_buffer managed by read and write indexes $readIndex\_A$, $writeIndex\_A$ and $readIndex\_B$, $writeIndex\_B$ respectively. An action is created just to read data from the FIFO and put it in the internal circular buffers while increment the read indexes. Later consuming the data from the buffers increments the write indexes. Consequently, the difference between the read and the write indexes represents the number of available tokens in each buffer and all the firing rules of the actions are related to this difference. An example of an internal buffer is shown in Figure 5.

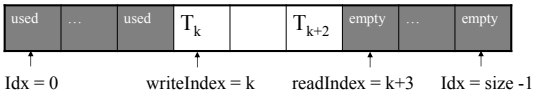

Fig. 5: Example of an internal buffer.

Using this methodology, the firing rule of equation 1 is transformed to the rule of equation 2:

$$\begin{cases} R_{1,1} = [readIndex\_A - writeIndex\_A >= 1] \\ R_{1,2} = [B\_buffer[writeIndex\_B] = 0, \\ \quad readIndex\_B - writeIndex\_B >= 1] \end{cases} \quad (2)$$

2) Scheduling translation: In order to have a correct hardware implementation it was imperative to update the actor scheduler to the Vivado HLS streams. Indeed, hardware FIFO are not unbounded. So while executing the actions that write tokens from internal circular buffers in the streams and the actions that pull tokens from the FIFO, respectively, it was necessary to ensure that the stream is not full, and not empty, respectively. As presented in Listing 3 that contains the c-HLS of the "add" actor, the add scheduler is updated to check the streams (not full or not empty) before the writing or reading data.

Listing 3: The c-HLS of the add actor.

```
#include <hls_stream.h>
using namespace hls;
typedef int i32;
extern stream<i32> stream_A;
extern stream<i32> stream_B;
extern stream<i32> stream_C;
static i32 B_buffer[1];
static i32 readIndex_B = 0;
static i32 writeIndex_B = 0;
static void read() {
        stream_A.read(a);
        stream_B.read(b);
}
bool read_isSchedulable(){
  return true;
}
static void add() {
        stream_C.write(a + b);
}
static bool add_isSchedulable() {
        bool result;
        b = B_buffer[0 + writeIndex_B & 0];
        result = b == 0;
        return result;
}
void add_scheduler() {
state_read:
  if(read_isSchedulable()){
    if(!stream_A.empty()&&!stream_B.empty()){
      read();
      goto state_add;
    }
  }else{
  goto state_read;
  }
state_add:
  if(add_isSchedulable()&&!stream_C.full()){
    add();
    goto state_read;
  }else{
    goto state_add;
  }
}
```

This code is composed of three main parts:

- streams and associated buffers and indexes declarations,
- actions called as functions followed by a Boolean function that returns the guard condition,
- global scheduler that checks the Boolean functions before firing the corresponding action. If there is an FSM, the scheduler manages the current and next state depending on which action is executed.

B. System level elaboration

In order to elaborate the system level, we take advantage from the fact that (1) Vivado HLS tool works well in generating hardware implementation from a unique actor and (2) the dataflow networks in RVC are described using XML derived languages as explained in Section II-B that can be parsed to extract actor connections information. In addition, while the streams are declared and used as externals making able the hardware component to communicate with the FIFO, the FIFO needs to be physically generated. For this issue, we modified

and used a generic FIFO component defined in the literature of Vivado HLS. The bit width of the FIFO is put as generic to match the bit width of the input and output data of the source and target actors and a "Top" VHDL file is automatically created to map the different signals of the components with the signals of the FIFO as illustrated in Figure 6.

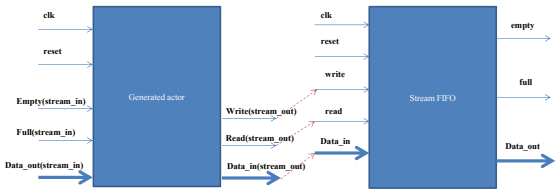

Fig. 6: Connections between the hardware FIFO and the generated actor as in the Top VHDL.

Thus, the system level is obtained and the different actors can fire in a parallel way. For a synchronous behavior, all clocks and reset signals are connected to those of the "Top" entity. For the validation of the generated design, we developed an automatic generation of test-benches for all granularity levels of the network which means that a test-bench is created for each actor, each network and each sub-network. This approach revealed to be very important to accelerate debugging and assessing the hardware generated implementation at component and system levels.

IV. Results

The purpose of this section is to show the performances of our proposed method for the automatic translation from dataflow programs to HDL by means of two design cases. The first is the MPEG-4 Simple Profile (SP) decoder, while the second is the HEVC intra-decoder. In the following we present an overview of these decoders' architecture and basic actors and, subsequently, the implementation results. The results presented are obtained using common simulators and synthesizers.

A. First case study: MPEG-4 SP

Figure 7 shows the MPEG-4 Part 2 SP decoder which has been described via a dataflow model using CAL and where the color space components Y, U, and V are decoded separately.

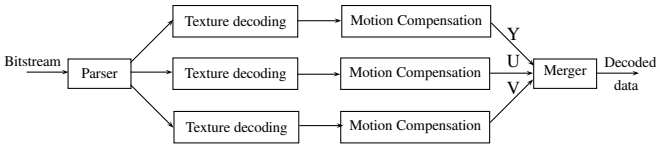

Fig. 7: MPEG-4 SP architecture.

The simulation results of the hardware implementation of the MPEG-4 SP decoder generated with the Vivado HLS tool are compared to those obtained with the Xronos tool. The simulated performance values are given in Table I for a stimulus frequency of 50 MHZ. Here, a Motion-MPEG stream consists of five QCIF images (176×144 pixels) has been used to obtain latency and throughput values.

TABLE I: MPEG-4 SP timing results.

	Xronos	Vivado HLS
Latency (ms)	0, 258	0, 158
Throughput (FPS)	232	125

After the simulation of the design, the HDL generated code was implemented on a Xilinx Virtex 4 platform (XC4VLX160). The area consumption results obtained are presented in Table II.

TABLE II: MPEG-4 SP area consumption (Virtex 4).

	Xronos	Vivado HLS
Slices	28823/67584(42%)	142302/67584(210%)
4 input LUTs	51898/135168(38%)	194583/135168(143%)
FIFO16/RAMB16s	223/288(77%)	150/288(52%)

Considering the comparison in Tables I and II, the Xronos design is found to be more efficient in terms of area consumption. This could be explained by the fact that the Xronos tool is a CAL dedicated tool. Despite the fact it uses more slices, the vivado HLS tool turns out to be a general tool adapted to any c-based descriptions. However, results presented on Table III highlights a profit in terms of area consumption for the design with the Vivado HLS tool on a recent Xilinx Virtex 7 platform (XC7V2000T).

TABLE III: MPEG-4 SP area consumption (Virtex 7).

	Vivado HLS
Number of Slice Registers	135503/2443200(5%)
Number of Slice LUTs	129403/1221600(10%)
Number of Block RAM/FIFO	75/1292(5%)

Nevertheless, concerning the time performances the proposed method reveal to be more efficient in terms of latency.

B. Second case study: HEVC

HEVC is currently being prepared as the newest video coding standard of the IUT-T Video Coding Experts Group and the ISO/IEC MPEG. The main goal of the HEVC standardization effort is to enable significantly improved compression performance relative to existing standards. Figure 8 gives an overview of the HEVC intra-decoder.

- The first part is the parser, an actor that extracts the different syntax elements of the different decoding stages from a bitstream.
- The IntraPrediction actor predicts the image blocks using neighbor blocks' values and a prediction mode information provided by the parser.
- The xIT actor is the integer transform that decodes the residual coefficients.
- The selectCU and LCU_reord are actors that reconstitute the image using the prediction mode information.

In the main design, there also the inter decoder part, the SAO and the deblocking filters, which would be dealt with in future

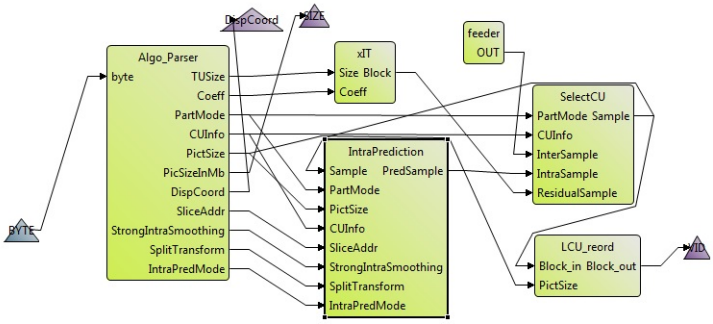

Fig. 8: HEVC intra-decoder architecture.

works.
Simulation results of our proposed method are shown in Table IV for a stimulus frequency of 20 MHZ. We use a sequence of 240p resolution under the Quantization Parameter (QP) value of 22.

TABLE IV: HEVC timing results.

	Vivado HLS
Latency	6, 68
Throughput (FPS)	6

The high latency of the HEVC intra-decoder was expected since the intra-prediction and the SelectCU actors store a big amount of tokens before starting the processes. The throughput frequency of 6 FPS can be far improved because there are several directives in the literature of Vivado HLS that will be exploited in the future work. We notice that this is a pioneer simulated hardware code generation of HEVC intra-decoder and the obtained results can be considered a starting point.

V. Conclusion and perspectives

This paper presents a new method which allows the implementation of a system operating at the highest level of abstraction compared to current approaches. It enables an automatic translation of dataflow programs written in actor/dataflow oriented language called CAL under the RVC standard into RTL descriptions. Such language has been specifically designed for modeling complex signal processing systems. Moreover, a development tool called Vivado HLS tool from Xilinx is used to convert C-based algorithms to hardware blocks. The functionality of the Vivado HLS tool was enhanced so it supports the entire system. The methodology used to adapt such tool to the constraints of RVC mainly the FIFO management was explained. Although our proposed method compared to existing approach seems to be less efficient in terms of area consumption, we effectively achieved a pioneer simulated hardware code generation of the most recent standard HEVC via our proposed design flow.

In future work, our goal is to improve performance and area using directives offered in the Vivado HLS tool such as binding the tool for more efficient generation, applying ping-pong buffers management and pipelining and paralleling functions. Moreover, in this work we used the streams to insure the communication between components, but it is possible to use directly Random Access Memory (RAM). In this case, a RAM would be accessible to spy tokens, thus there would be no need to create internal buffers. This perspective would have a very important impact on both the area and timing performances.

Acknowledgment

We would like to express our distinguished thanks to Hervé Yviquel for his contribution in C-HLS code to be compatible with the Vivado HLS tool. The authors are also particularly grateful to Gildas Cocherel for his important advises in the VHDL test-benches and streams communication.

References

[1] S. S. Bhattacharyya, J. Eker, J. W. Janneck, C. Lucarz, M. Mattavelli, and M. Raulet, "Overview of the MPEG Reconfigurable Video Coding Framework," *Journal of Signal Processing Systems*, vol. 63, no. 2, pp. 251–263, May 2011.

[2] J. Eker and J. W. Janneck, "Cal language report specification of the cal actor language," EECS Department, University of California, Berkeley, Tech. Rep. UCB/ERL M03/48, 2003.

[3] P. Coussy and A. Morawiec, *High-Level Synthesis From Algorithm to Digital Circuit*. Springer Publishing Company, 2008.

[4] G. Kahn, "The semantics of a simple language for parallel programming," *Information processing*, vol. 74, pp. 471–475, 1974.

[5] E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow," in *Proceedings of the IEEE*, vol. 75, no. 9, Sep. 1987, pp. 1235–1245.

[6] E. A. Lee and T. Parks, "Dataflow process networks," in *Proceedings of the IEEE*, 1995, pp. 773–799.

[7] G. Martin, B. Bailey, and A. Piziali, *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2007.

[8] J. W. Janneck, I. D. Miller, D. B. Parlour, G. Roquier, M. Wipliez, and M. Raulet, "Synthesizing hardware from dataflow programs: An MPEG-4 simple profile decoder case study," in *Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on*, Washington, États-Unis, 2008, pp. 287 – 292.

[9] T. Richard, R. Mosqueron, J. Dubois, and M. Mattavelli, "Generation of Hardware/Software systems based on CAL dataflow description," *lgorithm-Architecture Matching for Signal and Image Processing, Lectures Notes in Electrical Engineering*, vol. 73, no. 3, pp. 275–292, Jan. 2011.

[10] M. Wipliez, "Infrastructure de compilation pour des programmes flux de données," Ph.D. dissertation, INSA de Rennes, Dec. 2010.

[11] N. Siret, M. Wipliez, J. F. Nezan, and A. Rhatay, "Hardware code generation from dataflow programs," in *Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on*, Royaume-Uni, 2010, pp. 113 –120.

[12] S. S. Bhattacharyya, G. Brebner, J. W. Janneck, J. Eker, C. Von Platen, M. Mattavelli, and M. Raulet, "OpenDF – A Dataflow Toolset for Reconfigurable Hardware and Multicore Systems," in *First Swedish Workshop on Multi-Core Computing*, Ronneby, Suède, 2008, p. CD.

[13] E. Bezati, H. Yviquel, M. Raulet, and M. Mattavelli, "A unified hardware/software co-synthesis solution for signal processing systems," in *Design and Architectures for Signal and Image Processing (DASIP), 2011 Conference on*, France, 2011, pp. 1 –6.

[14] E. Bezati, R. Thavot, G. Roquier, and M. Mattavelli, "High-level dataflow design of signal processing systems for reconfigurable and multicore heterogeneous platforms," *Journal of Real-Time Image Processing*, pp. 1–12, 2013.

[15] K. Jerbi, M. Raulet, O. Déforges, and M. Abid, "Automatic Generation Of Optimized And Synthesizable Hardware Implementation From High-Level Dataflow Programs," *VLSI Design*, vol. 2012, p. Article ID 298396, 2012.