HAL
open science

Automatic Generation Of Optimized And Synthesizable Hardware Implementation From High-Level Dataflow Programs

Khaled Jerbi, Mickaël Raulet, Olivier Déforges, Mohamed Abid

To cite this version:

Khaled Jerbi, Mickaël Raulet, Olivier Déforges, Mohamed Abid. Automatic Generation Of Optimized And Synthesizable Hardware Implementation From High-Level Dataflow Programs. VLSI Design, 2012, 2012, pp.Article ID 298396. 10.1155/2012/298396 . hal-00724403

HAL Id: hal-00724403

https://hal.science/hal-00724403v1

Submitted on 20 Aug 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Hindawi Publishing Corporation
VLSI Design
Volume 2012, Article ID 298396, 14 pages
doi:10.1155/2012/298396

Research Article

Automatic Generation of Optimized and Synthesizable Hardware Implementation from High-Level Dataflow Programs

Khaled Jerbi,[1,2] Mickaël Raulet,[1] Olivier Déforges,[1] and Mohamed Abid[2]

[1] *IETR/INSA. UMR CNRS 6164, 35043 Rennes, France*
[2] *CES Laboratory, National Engineering School of Sfax, 3038 Sfax, Tunisia*

Correspondence should be addressed to Khaled Jerbi, khaled.jerbi@insa-rennes.fr

Received 16 December 2011; Revised 18 April 2012; Accepted 15 May 2012

Academic Editor: Maurizio Martina

Copyright © 2012 Khaled Jerbi et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

In this paper, we introduce the Reconfigurable Video Coding (RVC) standard based on the idea that video processing algorithms can be defined as a library of components that can be updated and standardized separately. MPEG RVC framework aims at providing a unified high-level specification of current MPEG coding technologies using a dataflow language called Cal Actor Language (CAL). CAL is associated with a set of tools to design dataflow applications and to generate hardware and software implementations. Before this work, the existing CAL hardware compilers did not support high-level features of the CAL. After presenting the main notions of the RVC standard, this paper introduces an automatic transformation process that analyses the non-compliant features and makes the required changes in the intermediate representation of the compiler while keeping the same behavior. Finally, the implementation results of the transformation on video and still image decoders are summarized. We show that the obtained results can largely satisfy the real time constraints for an embedded design on FPGA as we obtain a throughput of 73 FPS for MPEG 4 decoder and 34 FPS for coding and decoding process of the LAR coder using a video of CIF image size. This work resolves the main limitation of hardware generation from CAL designs.

1. Introduction

User requirements of high quality video are growing which causes a noteworthy increase in the complexity of the algorithms of video codecs. These algorithms have to be implemented on a target architecture that can be hardware or software. In 2007, the notion of Electronic System Level Design (ESLD) has been introduced in [1] as a solution to decrease the time to market using high-level synthesis which is an automatic compilation of high-level description into a low-level one called register transfer level (RTL). The high-level description is governed by models of computation which are the rules defining the way data is transferred and processed. Many solutions were developed to automate the hardware generation of complex algorithms using ESLD. Synopsys developed a C to gate compiler called synphony [2]. Mentor Graphics also created a C to HDL compiler called Catapult C [3, 4]. For their NIOS II, Altera introduces C2H as a converter from C to HDL [5, 6]. To extend Matlab for hardware generation from functional blocks, Mathworks created a hardware generator for FPGA design [7]. In the university research field, STICC laboratory in France developed a high-level synthesis tool called GAUT that extracts parallelism and generates VHDL code from a pure C description [8, 9]. The common point between all previously quoted tools is the fact that they are application-specific generators which means that they are not always efficient on an entire multi-component system description.

In this context, CAL [10] was introduced in the Ptolemy II project [11] as a general-use dataflow target agnostic language based on the dataflow Process Network (DPN) Model of Computation [12] related to the Kahn Process Network (KPN) [13]. The MPEG community standardized the RVC-CAL language in the MPEG RVC (Reconfigurable Video Coding) standard [14]. This standard provides a framework to describe the different functions of a codec as a network of functional blocks developed in RVC-CAL and called actors. Some hardware compilers of RVC-CAL were developed but their limitation is the fact that they cannot compile high-level

structures of the language so these structures have to be manually transformed.

In [15], we presented an original functional method to quicken the HDL generation using a software platform for rapid design and validation of a high complexity dataflow architecture but going from high to low-level representation used to be manual. Therefore, we proposed to add automatic transformations to make any RVC-CAL design synthesizable.

This paper extends a preliminary work presented in [16] by introducing efficient optimizations and their impact on the area and time consumption of the design. The transformation tool analyzes the RVC-CAL code and performs the required transformations to obtain synthesizable code whatever the complexity of the considered actor. In Section 2, we explain the main advantages of using MPEG RVC standard for signal processing algorithms and the key notions of the RVC-CAL language and its behavioral structures and mechanisms. The proposed transformation process is detailed in Section 4 and finally hardware implementation results of MPEG4 Part2 decoder and LAR codec are presented in Sections 5 and 6.

2. Background

Since the beginning of ISO/IEC/WG11 (MPEG) in 1988 with the appearance of MPEG-1, many video codecs have been developed (MPEG-4 part2, MPEG SVC, MPEG AVC, HEVC, etc.) with an increasing complexity and so they take longer time to be produced. In addition, every standard has a set of profiles depending on the implementation target or the user specifications. Consequently, it became a tough task for standard communities to develop, test, and standardize a decoder at any given time. Moreover, the standards specification is monolithic which makes it harder to reuse or update some existing algorithms. This ascertainment originated a new conception methodology standard called Reconfigurable Video Coding introduced by MPEG.

In the following, we present an overview of MPEG RVC standard and associated tools and frameworks, we also present the main features of CAL actor language and the limitations that motivated this work.

2.1. MPEG RVC. RVC presents a modular library of elementary components (actors). The most important and attractive features of RVC are reconfigurability and flexibility. An RVC design is a dataflow directed graph with actors as vertices and unidirectional FIFO channels as edges. An example of a graph is shown in Figure 1.

Actually, defining video processing algorithms using elementary components is very easy and rapid with RVC since every actor is completely independent from the rest of the other actors of the network. Every actor has its own scheduler, variables, and behavior. The only communication of an actor are its input ports connected to the FIFO channels to check the presence of tokens and as explained later an internal scheduler is going to allow or not the execution of elementary functions called actions depending on their corresponding firing rules (see Section 3). Thus, RVC insures concurrency, modularity, reuse, scalable parallelism, and encapsulation. In [17], Janneck et al. show that, for hardware designs, *RVC standard allows a gain of 75% of development time* and considerably reduces the number of lines compared with the manual HDL code. To manage all the presented concepts of the standard, RVC presents a framework based on the use of the following.

(i) A subset of the CAL actor language called RVC-CAL that describes the behavior of the actors (see details in Section 2.2).

(ii) A language describing the network called FNL (Functional unit Network Language) that lists the actors, the connections and the parameters of the network. FNL is an XML dialect that allows a multilevel description of actors hierarchy which means that a functional unit can be a composition of other functional units connected in another network.

(iii) Bitstream syntax Description Language (BSDL) [18, 19] to describe the structure of the bitstream.

(iv) An important Video Tool Library (VTL) of actors containing all MPEG standards. This VTL is under development and it already contains 3 profiles of MPEG 4 decoders (Simple Profile, Progressive High Profile and Constrained Baseline Profile).

(v) Tools for edition, simulation, validation and automatic generation of implementations:

(a) open DF framework [20] is an interpreter infrastructure that allows the simulation of hierarchical actors network. Xilinx contributed to the project by developing a hardware compiler called OpenForge (available at http://openforge.sourceforge.net/) [21] to generate HDL implementations from RVC-CAL designs.

(b) open RVC-CAL Compiler (Orcc) (available at http://orcc.sourceforge.net/) [19] is an RVC-CAL compiler under development. It compiles a network of actors and generates code for both hardware and software targets. Orcc is based on works on actors and actions analysis and synthesis [22, 23]. 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 that undergoes typing, semantic checks and several transformations in the middle-end and in the back-end. Finally, pretty printing is applied on the resulting IR to generate a chosen implementation language (C, Java, Xlim, LLVM, etc.).

At this level, the question is that *why RVC-CAL and not C*? Actually, a C description involves not only the specification of the algorithms but also the way inherently parallel computations are sequenced, the way data is exchanged throw inputs and outputs, and the way computations are mapped. Recovering the original intrinsic properties of the algorithms by analyzing the software program is impossible.

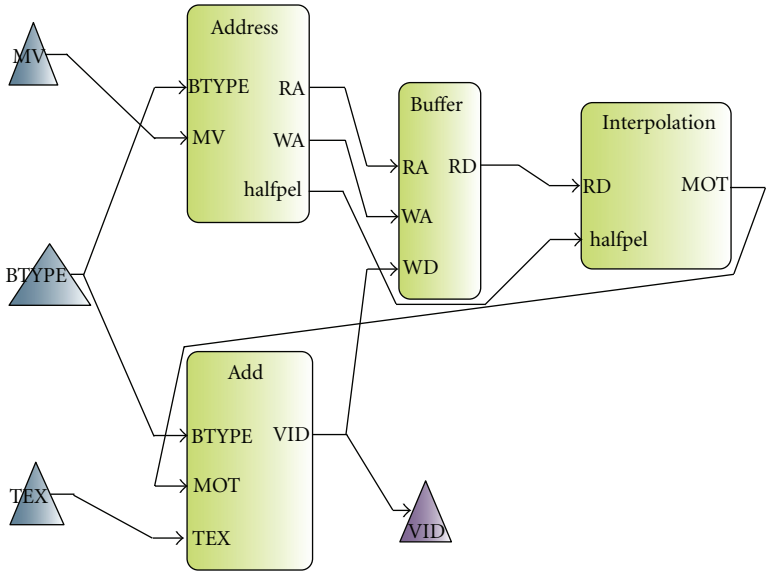

Figure 1: Graph example. Block diagram of the motion compensation of an MPEG 4 part 2 decoder.

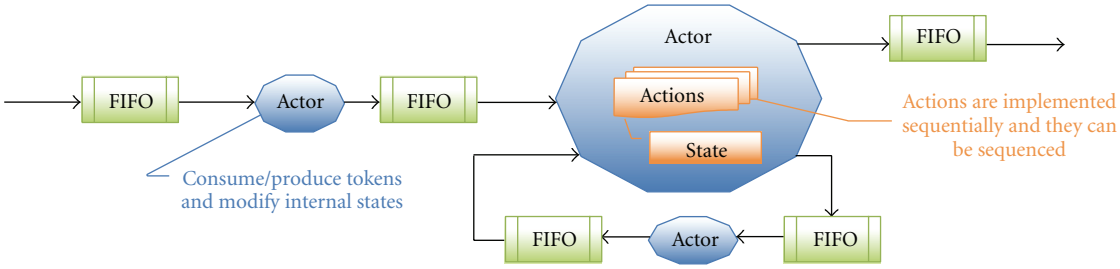

Figure 2: CAL actor model.

In addition, the opportunities for restructuring transformations on imperative sequential code are very limited compared to the parallelization potential available on multi-core platforms. For these reasons, RVC adopted the CAL language for actors specification. The main notions of this language are presented below.

2.2. CAL Actor Language. The execution of an RVC-CAL code is based on the exchange of data tokens between computational entities (actors). Each actor is independent from the others since it has its own parameters and finite state machine if needed. Actors are connected to form an application or a design, this connection is insured by FIFO channels. Executing an actor is based on *firing* elementary functions called *actions*. This action firing may change the state of the actor in case of an FSM. An RVC-CAL dataflow model is shown in the network of Figure 2.

Figure 3 presents an example of a CAL actor realizing the sum between two tokens read from its two input ports.

Like in VHDL, an actor definition begins by defining the I/O ports and their types then actions are later listed. An action begins also by defining the I/O ports it uses from the list of ports of the actor and this definition includes the number of tokens this action have to find in the FIFO to be fireable. In the "sum" actor, the internal scheduler allows action "add" only when there is at least one token in the FIFO of port "INPUT1" and one token in the FIFO of port "INPUT2" and this property explains how an actor can be totally independent and can neither read nor modify the state of any other actor. Of course, an actor may contain any number of actions that can be governed by an internal finite state machine. At a specific time two or more actions may have the required conditions to be fired so the notion of priority was introduced (see details in Section 3).

For the same behavior, an actor may be defined in different ways. Let us consider the "sum-5" actor of Figure 4 that reads 5 tokens in a port "IN," computes their sum and produces the result in a port "OUT."

In Figure 4(a), the required algorithm is defined in only one action. The condition of 5 required tokens is expressed by the instruction "repeat 5." Action "add" fires by consuming the 5 tokens from the FIFO into an internal buffer "I." After data storage, the algorithm of the action is applied. Finally the action firing finishes by writing the result in the port "OUT".

Such description is very fast to develop and implement on software targets but for hardware implementations a multitoken read is not appropriate. This is the reason of

```
actor sum ()
(int size=8) INPUT1, (int size=8) INPUT2 ==> int(size=8) OUTPUT:

 add: action INPUT1:[ i1 ], INPUT2[i2] ==> OUTPUT:[s]
 var
   int s
 do
   s:= i1 + i2 ;
 end
end
```

Figure 3: Example of sum actor.

```
actor sum-5 () int (size=8) IN
==> int(size=8) OUT:

 add: action IN:[ i ] repeat 5
 ==> OUT:[ s ]
 var
   int s := 0
 do
   foreach int k in 0 .. 4 do
     s := s + i[k] ;
   end
 end
end
```

(a) SW-oriented definition

```
actor sum-5 () int (size=8) IN
==> int(size=8) OUT:

 List (type: int (size=8), size = 5) data;
 int counter :=0 ;

 read: action IN:[ i ] ==>
 do
   data[counter] := i ;
   counter := counter + 1 ;
 end

 read_done: action ==>
 guard
   counter = 5
 do
   counter := 0 ;
 end

 process: action ==> OUT:[ s ]
 var
  int s := 0
 do
   foreach int k in 0 .. 4 do
     s := s + data[k] ;
   end
 end

 schedule fsm state0:
   state0 (read)      --> state0;
   state0 (read_done) --> state1;
   state1 (process)   --> state0;
 end

 priority
   read_done > read;
 end

end
```

(b) HW-oriented definition

Figure 4: Two-way definition example of sum-5 actor behavior.

developing the equivalent monotoken code of Figure 4(b). In this description, we use a finite state machine to lock the actor in the state "state0." While counter¡5, only the action "read" can be fired to store tokens one per one in "data" buffer. Once the condition of action "read_done" (counter = 5) is true, both of "read" and "read_done" actions are fireable. This is why the priority "read_done ≻ read" is important to keep the determinism of the actor. Finally, the firing of "read_done" action involves an FSM update to "state1" where only "process" action can be fired and the actor is back to the initial state.

3. Actor Behavior Formalism

Actor execution is governed by a set of conditions called firing rules. Moreover, during this firing many internal features of the actor are updated (state, state variables, etc.). All these concepts and behavior evolutions are detailed below. The actor execution, so called firing, is based on the dataflow Process Network (DPN) principle [12] derived from the Kahn Process Network (KPN) [13]. Let Ω be the universe of all tokens values exchanged by the actors and $\mathbb{S} = \Omega^*$, the set of all finite sequences in Ω. We denote the length of a sequence $s \in \mathbb{S}^k$ by $|s|$ and the empty sequence by λ. Considering an actor with m inputs and n outputs, $\mathbb{S}^m$ and $\mathbb{S}^n$ are the set of m-tuples and n-tuples consumed and produced. For example, $s_0 = [\lambda[t_0, t_1, t_2]]$ and $s_1 = [[t_0], [t_1]]$ are sequences of tokens that belong to $\mathbb{S}^2$ and we have $|s_0| = [0, 3]$ and $|s_1| = [1, 1]$.

3.1. Actor Firing. A dataflow actor is defined with a pair $\langle f, R\rangle$ such as:

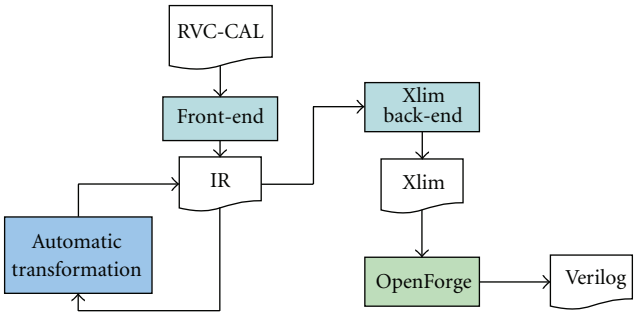

Figure 5: Automatic transformation localization in Orcc compiling process.

(i) $f : \mathbb{S}^m \rightarrow \mathbb{S}^n$ is the firing function;

(ii) $R \subset \mathbb{S}^m$ are the firing rules;

(iii) for all $r \in R$, $f(r)$ is finite.

An actor may have N firing rules which are finite sequences of m patterns (one for each input port). A pattern is an acceptable sequence of tokens for an input port. It defines the nature and the number of tokens necessary for the execution of at least one action. RVC-CAL also introduces the notion of *guard* as additional conditions on tokens values. An example of firing rule r_j in $\mathbb{S}^2$ is

$$\begin{aligned} g_{j,k} &: [x] \mid x > 0 \\ r_j &= \left[t_0 \in g_{j,k}, [t_1, t_2, t_3]\right], \end{aligned} \tag{1}$$

Equation (1) means that if there is a positive token in the FIFO of the first input port and 3 tokens in the FIFO of the second input port then the actor will select and execute a fireable action. An action is fireable or schedulable iff:

(i) the execution is possible in the current state of the FSM (if an FSM exists);

(ii) there are enough tokens in the input FIFO;

(iii) a guard condition returns true.

An action may be included in a finite state machine or untagged making it higher priority than FSM actions.

3.2. Actor Transition. The FSM transition system of an actor is defined with $\langle \sigma_0, \Sigma, \tau, \prec \rangle$ where Σ is the set of all the states of the actor, σ_0 is the initial state, $\prec$ is a priority relation and $\tau \subseteq \Sigma \times \mathbb{S}^m \times \mathbb{S}^n \times \Sigma$ is the set of all possible transitions. A transition from a state σ to a state σ' with a consumption of sequence $s \in \mathbb{S}^m$ and a produced sequence $s' \in \mathbb{S}^n$ is defined with (σ, s, s', σ') and denoted.

$$\sigma \xrightarrow[\tau]{s \mapsto s'} \sigma'. \tag{2}$$

To solve the problem of the existence of more than one possible transition in the same state, RVC-CAL introduced the notion of priority relation such as for the transitions $t_0, t_1 \in \tau$, t_0 a higher priority than t_1 is written $t_0 \succ t_1$. As explained in [24] a transition $\sigma \xrightarrow[\tau]{s \mapsto s'} \sigma'$ is enabled iff:

$$\neg \exists \sigma \xrightarrow[\tau]{p \mapsto q} \sigma'' \in \tau : p \in \mathbb{S} \wedge \sigma \xrightarrow[\tau]{s \mapsto s'} \sigma'' \succ \sigma \xrightarrow[\tau]{s \mapsto s'} \sigma'. \tag{3}$$

This section presented and explained the main RVC-CAL principles. In the next section we present an automatic transformation as a solution to avoid these limitations without changing the overall macrobehavior of the actor.

3.3. Hardware Generation Problematic. A firing rule is called *multitoken* iff : $\exists e \in |s| : e > 1$ otherwise it is called a *monotoken* rule. The limitation of OpenForge is the fact that it does not support multitoken rules which are omnipresent in most actors. The observation of Figure 4 shows the incontestable complexity difference between the multitoken (a) and the monotoken (b) code. Moreover, manually changing a CAL code from high-level to low-level by creating the new actions, variables and state machine is contradictory to the main purpose of RVC standard which is the fact that CAL is a target agnostic language so we have to write in CAL the same way for hardware of software implementation. Our work consists in automatically transforming the data read/write processes from multitoken to monotoken while preserving the same actor behavior. All the required actions, variables and finite state machines are created and optimized directly in the Intermediate representation of Orcc compiler. The following section explains the achieved transformation mechanism.

4. Methodology for Hardware Code Generation

As shown in Figure 5, our transformation acts on the IR of Orcc. The HDL implementation is later generated using OpenForge.

4.1. Actor Transformation Principle. Let us consider an actor with a multitoken firing rule $r \in \mathbb{S}^k$ such as $|r| = [r_0, r_1, \ldots, r_{k-1}]$, this rule fires a multitoken action a realizing the transition $source \xrightarrow[\tau]{a} target$ and $\mathbb{I}$ the set of all input ports. The transformation creates for every input port an internal buffer with read-and-write indexes and clips r into a set $\mathbb{R}$ of k firing rules so that:

$$\forall i \in \mathbb{I}, \exists! \rho \in \mathbb{R} : \begin{cases} \rho : \mathbb{S}^1 \longrightarrow \mathbb{S}^0 \\ |r| = 1 \\ g_\rho : IdxWrite_i - IdxRead_i \leq sz_i, \end{cases} \tag{4}$$

with ρ a monotoken firing rule of an untagged action $untagged_i$, g_ρ is the guard of ρ, and sz_i the size of the associated internal buffer defined as the closest power of 2 of r_i. This guard checks that the buffer contains an empty place for the token to read. The multitoken action is consequently removed, and new *read actions* that read one token from the internal buffers are created. While reading tokens another firing rule may be validated and causes the firing of an unwanted action. To avoid the nondeterminism of such a case, we use an FSM to put the actor in a reading loop so it can only read tokens. The loop is entered using a *transition*

```
actor A () int IN1, int IN2, int IN3 ==> int OUT1, int OUT2:
   a: action
   in1:[in1] repeat 2, IN2:[in2] repeat 3, IN3:[in3] ==>
   OUT1:[out1], OUT2:[out2] repeat 2
   do
      {treatment}
   end
end
```

Figure 6: RVC-CAL code of actor A.

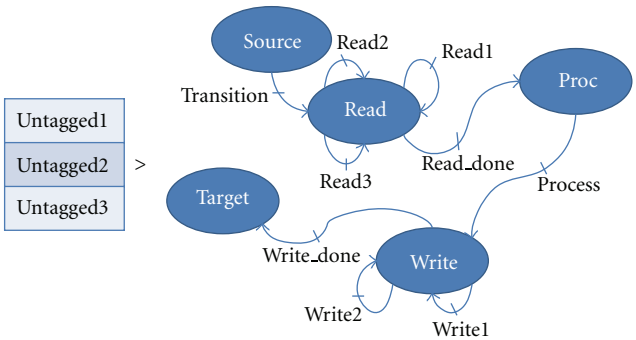

Figure 7: Created FSM macroblock.

action realizing the FSM passage $source \xrightarrow[\tau]{transition} read$ and has the same priority order of the deleted multitoken action but has no process. The read actions loop in the read state with the transition $t = read \xrightarrow[\tau]{read} read$. Then the loop is exited when all necessary tokens are read using a *read done* action and a transition to the process state $t' = read \xrightarrow[\tau]{read\ Done} process \succ t$. The treatment of the multitoken action is put in a *process action* with a transition $process \xrightarrow[\tau]{process} write$. The multitoken outputs are also transformed into a writing loop with *write actions* that store data directly in the output FIFO associated with a transition $w = write \xrightarrow[\tau]{write} write$ and a *write done* action that insures the FSM transition $w' = write \xrightarrow[\tau]{write\ Done} target \succ w$.

For example, the actor A of Figure 6 is defined with $f{:}\mathbb{S}^3 \to \mathbb{S}^2$ with a multitoken firing rule: $r \in \mathbb{S}^3 : r = [[t_0, t_1], [t_2, t_3, t_4], [t_5]]$.

Consequently, $|r| = [2, 3, 1]$ which means that there is an action in A that fires if 2 tokens are present in $IN1$ port, 3 tokens are present in $IN2$ and one token is present in $IN3$. The transformation creates the FSM macroblock of Figure 7.

4.2. FSM Creation Cases. We consider an example of an actor defined as $f : \mathbb{S}^3 \to \mathbb{S}^2$ containing the actions $a1 \cdots a5$ such as $a3$ is the only action applying a multitoken firing rule $r \in \mathbb{S}^3$.

Creating an FSM only for action $a3$ is not appropriate because $a1, a2, a4, a5$ will be a higher priority which may not be true. The solution is to create an initial state containing all the actions and add the created FSM macroblock of $a3$ (previously presented in Figure 7). The resulting FSM is presented in Figure 8.

We now suppose the same actor scheduled with an initial FSM as shown in Figure 9.

The transition $t = S1 \xrightarrow[\tau]{s \mapsto s'} S2$ is substituted with the macroblock of $a3$ as shown in Figure 10.

4.3. Optimizations. To improve the transformation, some optimization solutions were added. In the previously presented transformation method we used the untagged actions to store data in the internal buffers, then we used read actions to peek the required tokens from the internal buffers using R/W indexes and masks. To preserve the schedulability, the action is split into a transition action that contains the firing rule and a process action that applies the algorithm. The proposed optimization consists in making the action reading directly from the internal buffers. The firing rule of the action is transformed as presented in (4) to detect the presence of enough data in the internal buffers. Let us reconsider the basic example of the "sum-5" actor of Figure 4 of Section 2.2. The transformation explained above and the optimized transformation of this actor are presented in Figure 11. This actor is transformed this way. First an internal buffer and an untagged action are created to store data inside the actor. The input pattern of the *read* action is transformed into a connection to the internal buffer. Every read or write from the internal buffer must be masked to make the modulo of th buffer size since it is circular.

5. RVC Case of Study: MPEG 4 SP Intradecoder

To assess the performance of the previously presented transformation, we applied it on the whole MPEG 4 simple profile intradecoder. This choice is explained by the fact that there exists a stable design in the VTL and also because this decoder includes various image processing algorithms with more or less complexity. In the following we present an overview of this codec architecture and basic actors. We also present the implementation results and a comparison with an academic high-level synthesis tool called GAUT.

5.1. Concept. MPEG codecs have all a common design. It begins with a parser that extracts motion compensation and texture reconstruction data. The parser is then followed by reconstruction blocs for texture and motion and a merger as presented in Figure 12. This decoder is a full example of coding techniques that encapsulates predictions, scan, quantization, IDCT transform, buffering, interpolation, merging and especially the very complex step of parsing.

Table 1 gives an idea about the complexity of parsers in MPEG 4 Simple Profile and MPEG Advanced Video Coding (AVC).

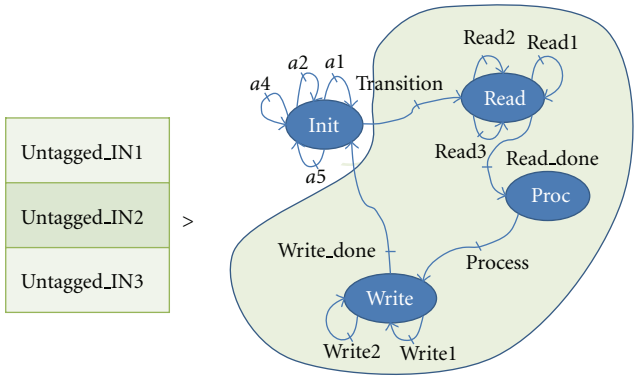

Figure 8: FSM with created initial state.

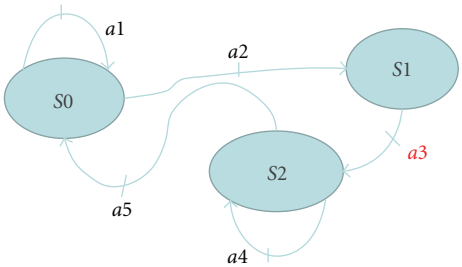

Figure 9: Initial FSM of an actor.

Table 1: Composition of MPEG-4 simple profile and MPEG-4 advanced video coding RVC-CAL description.

	Actors	Levels	Parser size kSLOC	Decoder size kSLOC
MPEG-4 SP	27	3	9.6	2.9
MPEG-4 AVC	45	6	19.8	3.9

Table 2: MPEG4 decoder area consumption.

Criterion	Transformed design	Optimized design
Slice flip flops	21,624/135,168 (15%)	13,575/135,168 (10%)
Occupied slices	45,574/67,584 (67%)	18,178/67,584 (26%)
4 input LUTs	68,962/135,168 (51%)	34,333/135,168 (25%)
FIFO16/RAMB16s	14/288 (4%)	14/288 (4%)
Bonded IOBs	107/768 (13%)	107/768 (13%)

Actors of Figure 12 are the main functional units some of them are hierarchical composition of actor networks. An actor may be instantiated more than one time so for 27 FU there are 42 actor instantiations.

5.2. Implementation and Results. The achieved automatic transformation was applied on MPEG4 SP intradecoder (see design in Orcc Applications (available at http://orcc.sourceforge.net/)) which contains 29 actors. We omitted the inter decoder part because it is very memory consuming. The HDL generated code was implemented on a virtex4 (xc4vlx160-12ff1148) and the area consumption results we obtained are presented in Table 2. The removal of read actions buffers and process actions had an important impact on the area consumption since it has decreased about 50%.

Table 3: MPEG4 decoder timing results.

Criterion	Transformed design	Optimized design
Maximum frequency (MHz)	26.4	26.67
Latency (μs)	381.8	306.4
Cadency (MHz)	1.9	2.33
Processing time (ms/image)	13.55	11.01
Throughput frequency (MHz)	1.8	2.2
Global image processing (FPS)	73.8	90.82

After the synthesis of the design, we applied a simulation stream of compressed videos. Table 3 below presents the timing results of a CIF (352×288) image size video.

We notice that timing results were partially improved. This is due to the presence of division operations in some actors. In our transformation we replaced divisions by an Euclidean division which is very costly and time consuming. The impact is noticeable since these divisions reduced the maximum frequency by 60%. Therefore, we applied the transformation on the inverse discrete cosine 2D transform (IDCT2D). We chose this actor because it contains very complex algorithm, functions and procedures. We tried to compare with an optimal low-level architecture designed by Xilinx experts and also with an existing implementation study of a direct VHDL written algorithm in [25]. For a significant comparison, we used the same implementation target

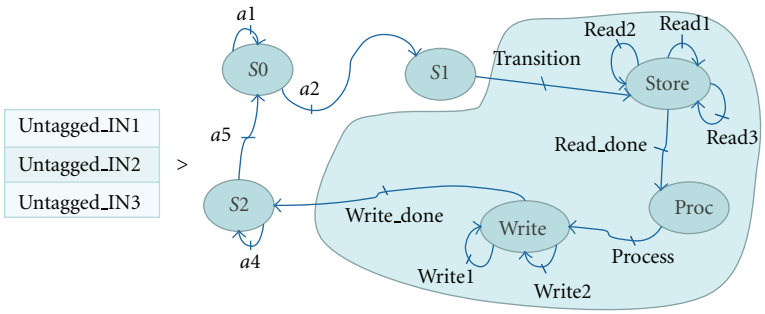

Figure 10: Resulting FSM transformation.

```
actor sum-5 () int (size=8) IN
==> int(size=8) OUT:

 List (type: int (size=8), size = 8) buffer;
 // closest power of 2 for circular buffer
 List (type: int (size=8), size = 5) data;
 int readIdx := 0;
 int writeIdx := 0;
 int counter :=0 ;

 action IN:[ i ] ==> // untagged action
 guard
   readIdx - writeIdx < 8
   // condition that the buffer is not full
 do
   buffer[readIdx & 7] := i ;
   // masked read index
   readIdx := readIdx + 1 ;
 end

 read: action ==>
 do
   data[counter] := buffer[writeIdx & 7] ;
   // masked write index
   counter := counter + 1 ;
 end

 read_done: action ==>
 guard
   counter = 5
 do
   counter := 0 ;
 end

 process: action ==> OUT:[ s ]
 var
  int s := 0
 do
   foreach int k in 0 .. 4 do
     s := s + data[k] ;
   end
   writeIdx := writeIdx + 5; // update writeIdx
 end

 schedule fsm state0:
   state0 (read)      --> state0;
   state0 (read_done) --> state1;
   state1 (process)   --> state0;
 end

 priority
   read_done > read;
 end

end
```

(a) Transformed equivalent CAL

```
actor sum-5 () int (size=8) IN
==> int(size=8) OUT:

 List (type: int (size=8), size = 8) buffer;
 int readIdx := 0;
 int writeIdx := 0;

 action IN:[ i ] ==>
 guard
   readIdx - writeIdx < 8
 do
   buffer[readIdx & 7] := i ;
   readIdx := readIdx + 1 ;
 end

 process: action ==> OUT:[ s ]
 guard
        readIdx - writeIdx > 5
 var
  int s := 0
 do
   foreach int k in 0 .. 4 do
     s := s + buffer[k + (writeIdx&7)] ;
   end
   writeIdx := writeIdx + 5; // update writeIdx
 end

end
```

(b) Optimized equivalent CAL

Figure 11: Transformed and optimized sum-5 actor.

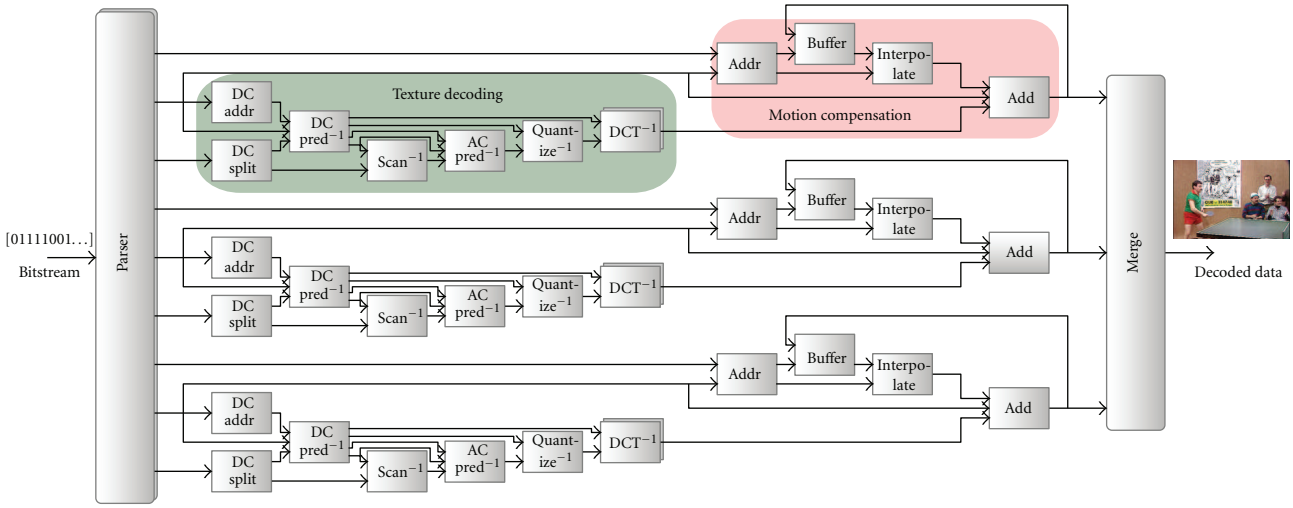

Figure 12: MPEG 4 SP architecture.

Table 4: IDCT2D timing results.

Image size	Xilinx design	Transformed design	Optimized design	VHDL design
Maximum frequency (MHz)	37	37	43	41
Latency (μs)	11.52	82.7	28.4	*
Cadency (MHz)	30	18.49	21.7	71
Processing time (μs/64 Tokens)	1.99	3.4	2.8	0.89
Throughput frequency (MHz)	26.62	0.72	2.43	62.4
Global image processing (FPS)	1064	31	101	2518

*Not mentioned in the literature.

of the study which is the Xilinx Spartan 3 XC3S4000. Timing and area consumption results comparison are presented in Tables 4 and 5.

Obviously, Table 5 reveals that area results for the optimized design are very close to those of the Xilinx low-level design. This property is noted for all actors containing more computing algorithms then data control and management algorithms. Concerning the area consumption of the VHDL design, it is expectable to find results nearby the optimal design and clearly worse than the Xilinx design and this is due to the synthesis constraints indicated in [25] that favor treatment speed in spite of the surface. This is what explains also the very high FPS rate of the design presented in Table 4. Timing results of the other designs show that the optimized design performances are far from the optimal Xilinx design. This is due to the low level architecture made by Xilinx experts which is completely different and oriented for hardware generation. This architecture is a pipelined set of actors realizing the IDCT2D (rowsort, fairmerge, IDCT1D, separate, transpose, retranspose, and clip) which is a relatively complex design compared with the high-level IDCT2D code used for the transformation.

After comparing with the Xilinx design and a VHDL directly written design, we compared our results with existing generation tools and we considered GAUT hardware generator. This tool is an academic high-level synthesizer from C to VHDL. It extracts the parallelism and creates a scheduled dependency graph made of elementary operators. Potentially, GAUT synthesizes a pipe-lined design with memory unit, communication interface and a processing unit. However, like most existing hardware generators, GAUT is not able to manage a system level design with very high complexity and a variety of processing algorithms. Moreover, there are so many restrictions on the C input code to have a functioning design. As it was impossible to test the whole MPEG 4 decoder we chose the IDCT2D algorithm to have a comparison with previously presented results.

The IDCT2D is so generated with GAUT and we obtained the results of Table 6 below.

Results show that the optimized transformation generates a better design even for the specific case of study of the IDCT2D.

6. Still Image Codec: LAR Case of Study

The LAR is a still-image coder [26] developed at the IETR/INSA of Rennes laboratory. It is based on the idea that the spatial coding can be locally dependent on the activity in the image. Thus, the higher the activity the lower the resolution is. This activity is dependent from the variation or the uniformity of the local luminance which can be detected using a morphological gradient. In the following, we detail

Table 5: IDCT2D area consumption.

Criterion	Xilinx design	Transformed design	Optimized design	VHDL design
Slice flip flops	1415/55296 (2%)	4002/55296 (7%)	2113/55296 (3%)	*
Occupied slices	1308/27648 (4%)	5238/27648 (18%)	2523/27648 (9%)	3571/27648 (12%)
4 input LUTs	2260/55296 (4%)	9861/55296 (17%)	4777/55296 (8%)	4640/55296 (8%)
Bonded IOBs	48/489 (9%)	49/489 (10%)	49/489 (10%)	*

*Not mentioned in the literature.

Table 6: IDCT2D area consumption with GAUT.

Criterion	GAUT design	Optimized design
Slice flip flops	2.080/135.168 (2%)	1.988/135.168 (2%)
Occupied slices	2.477/67.584 (3%)	2.353/67.584 (3%)
4 input LUTs	4.243/135.168 (3%)	4.458/135.168 (3%)
Bonded IOBs	627/768 (81%)	49/768 (6%)

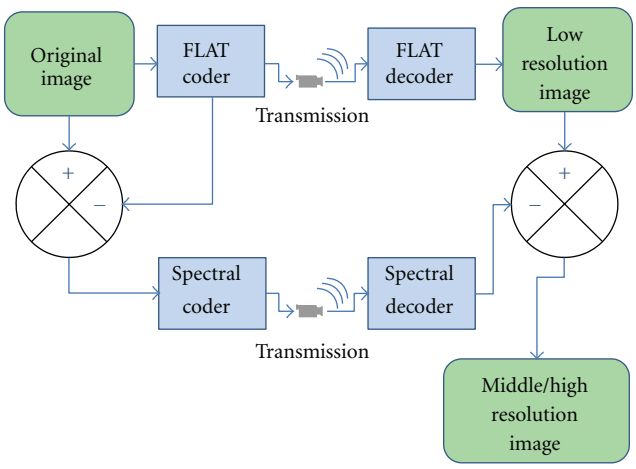

Figure 13: LAR concept.

coding principle of the LAR and we present the implementation techniques and results using the automatic transformation approach.

6.1. Concept. The LAR coding is based on considering that an image is a superposition of a global information image (mean blocks image), and the local texture image, which is given by the difference between the original image and the global one. This principle is modeled by

$$I = \bar{I} + \left(\underbrace{I - \bar{I}}_{E}\right), \tag{5}$$

where I is the original image, $\bar{I}$ is the global information image and $I - \bar{I}$ is the error image, E. The dynamic range of the error image is consequently dependent on the local activity. In uniform regions, $\bar{I}$ values are close or equal to I consequently $I - \bar{I}$ values are around zero with a low dynamic range.

Considering these principles, the LAR coder concept (Figure 13) is composed of two parts: the FLAT LAR [27] which is the part insuring the global information coding and the spectral part which is the error spectral coder.

Different profiles have been designed to fit with different types of application. In this paper, we focus on the baseline coder. Its mechanisms are detailed in the following.

The FLAT LAR. The Flat LAR is composed of 3 main parts: the partitioning, the block mean value computation and the DPCM (Differential Pulse Coding Modulation). In our work, only the DPCM is not yet developed with RVC-CAL.

(i) Partitioning: in this part, a Quad-tree partitioning is applied on the image pixels. The principle is to consider the lowest block size (2×2) then to compare the difference between the maximum (MAX) and the minimum (MIN) values of the block with a threshold (THD) defined as a generic variable for the design. If (MAX−MIN) > THD then the actual block size is considered. In the other case, the $(N\times2)\times(N\times2)$ size block is required. this process is recursively applied on the whole image blocks. The output of the overall is the block size image.

(ii) Block mean values computation process: this process is based on the Quad-tree output image. For each block of the variable size image, a mean value is put in the block as presented in the example of Figure 14.

(iii) The DPCM: the DPCM process is based on the prediction of neighbor values and the quantization of the block mean value image. The observation that a pixel value is mostly equal to a neighbor one led to the following estimation algorithm. If we consider the pixels in Figure 15, X value is estimated with the following algorithm:

If $|B - C| < |A - B|$ then $X = A$ else $X = C$.

The spectral coder, also called the texture coder, is composed of a variable block size Hadamard transform [28] and the Golomb-Rice [29, 30] entropy coder. The Golomb-Rice coder is still in development with the RVC-CAL specifications.

The Hadamard transform derives from a generalized class of the Fourier transform. It consists of a multiplication of a $2^m \times 2^m$ matrix by an Hadamard matrix (H_m) that has the same size. The transform is defined as follows.

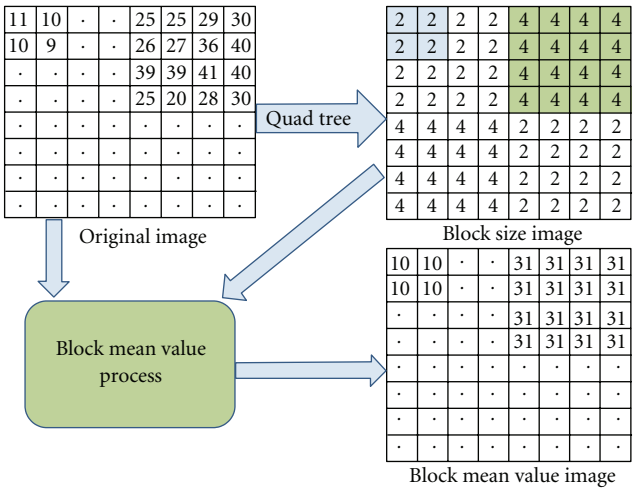

Figure 14: Block mean value process example.

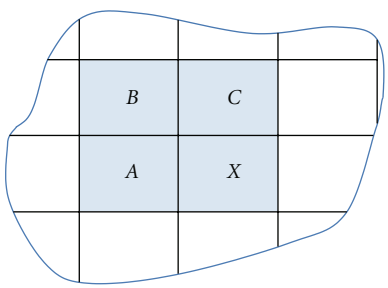

Figure 15: DPCM prediction of neighbor pixels.

H_0 is the identity matrix so $H_0 = 1$. For any $m > 0$, H_m is then deducted recursively by:

$$H_m = \frac{1}{\sqrt{2}} \begin{vmatrix} H_{m-1} & H_{m-1} \\ H_{m-1} & -H_{m-1} \end{vmatrix}. \tag{6}$$

Here are examples of Hadamard matrices:

$$H_0 = 1,$$
$$H_1 = \frac{1}{\sqrt{2}} \begin{vmatrix} 1 & 1 \\ 1 & -1 \end{vmatrix},$$
$$H_2 = \frac{1}{\sqrt{2}} \begin{vmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{vmatrix}, \quad \text{and so forth.} \tag{7}$$

6.2. Implementation and Results. This Section explains the mechanisms of the Hadamard transform and the Quad-tree used in the implementation.

6.2.1. Hardware Implementation. The LAR coding is dependent from the content of the image. It applies in the Quad-Tree a morphological gradient to extract information about the local activity on the image. The output is the block size image represented by variable size blocks: 2×2, 4×4 or 8×8. Using the block size image, the Hadamard transform applies the adequate transform on the corresponding block. It means that if we have a block size of 2×2 in the size image this block will undergo a 2×2 Hadamard (H_1) and a normalization specific to the 2×2 blocks. This process is identically applied for 4×4 and 8×8 blocks. A quantization step, adapted to current block size, is applied on the Hadamard output image. For each block size, a quantization matrix is predefined. Practically, the normalization during the Hadamard transform is postponed to be achieved with the quantization step so that to decrease the noise due to successive divisions.

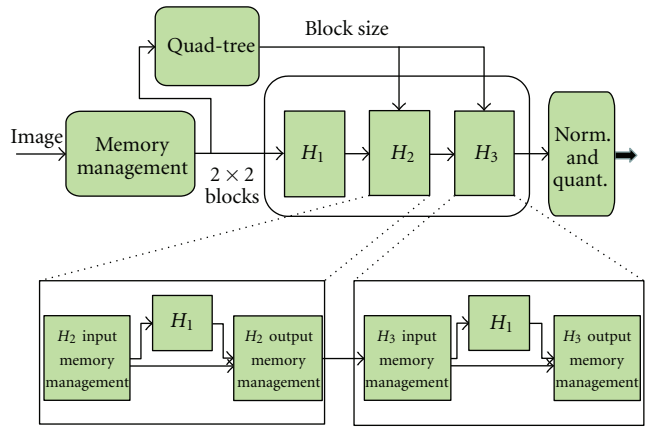

Figure 16: LAR baseline developed model.

The implemented LAR is presented in Figure 16.

As a first step, the memory management block stores the pixels values of the original image line by line. Once an 8×8 block is obtained, the actor divides it into sixteen 2×2 blocks and sends them in a specific order as presented in Figure 18.

This order is very important to improve the performance of remaining actors. In fact, considering the Figure 18, when the tokens are so ordered the first 4 tokens correspond to the first 2×2 block, the first 16 tokens to the first 4×4 block, and so forth Consequently, and as presented in Figure 16, the output of the H_1 is automatically the input of the H_2 and the output of the H_2 is automatically the input of the H_3.

In the Quad-tree, this order is also crucial. As presented in Figure 17, the superposition of the same actor (max for example) three times provides in the output of the first actor the maximum values of 2×2 blocks, in the output of the second actor the maximum values of 4×4 block and finally the maximum values of 8×8 blocks in the output of the third one. Using the maximum values and the minimums the morphological gradient in the Gradstep actors can process to extract the block size image. The same tip is used to calculate the block sums with three superposed sum actors. The block mean value actor considers the sums and the sizes to build the block mean value image.

We also notice that an (H_2) transform can be achieved using the (H_1) results of the four 2×2 blocks constituting the 4×4 block. The same observation can be made for the (H_3) one. This ascertainment is very important to decrease the complexity of the process. In fact, the Hadamard transform

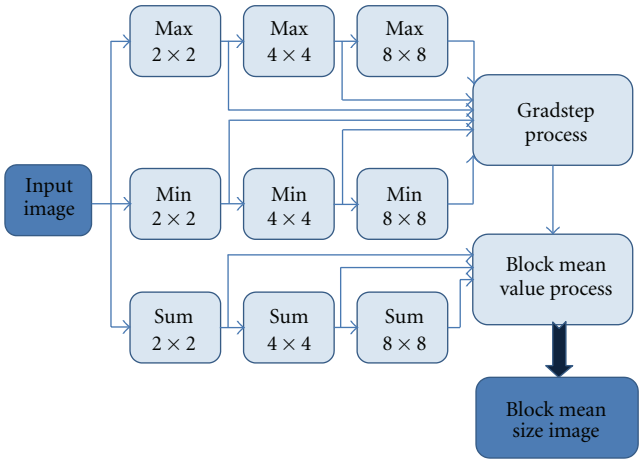

Figure 17: Quad-tree design.

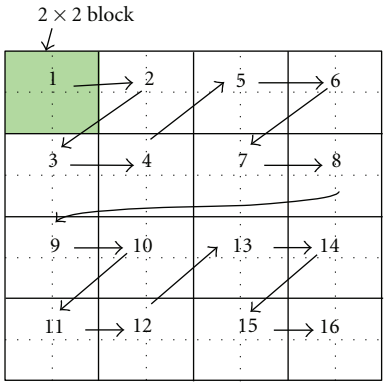

Figure 18: Memory management unit output order.

of the LAR applies an (H_1) transform for the whole image then it applies the (H_2) transform only for the 4×4 and 8×8 blocks and the (H_3) transform only for the 8×8 blocks. The (H_2) and the (H_3) transforms are different from the full transforms as they are much less complex. Consequently, as shown in Figure 16, we designed the H_2 and the H_3 using H_1 actors associated with memory management units. They sort tokens in the adequate order and, considering the block size, whether the block is going to undergo the transform or not.

It is very important to mention that almost actors have been developed with generic variables for memory sizes or gradsteps which means that the design are flexible for easy transformation from an image size to another or for adding higher Hadamard process (H_4, H_5, etc.).

In [15], we added some optimizations on the processes using a Ping-Pong memory management algorithm [31] to pipeline the process.

6.2.2. Results and Comparison. As mentioned above, this work aims at comparing hardware implementation performances of the same LAR architecture generated with the optimized automatic transformation and with a manual transformation. The achieved automatic transformation was applied on the 23 actors of the LAR using Orcc. The HDL generated code was implemented on a virtex4 (xc4vlx160-12ff1148). The area consumption results obtained are presented with those of manual transformations in Table 7.

Table 7: LAR coder area consumption.

Transformation	Automatic	Manual
Slice flip flops	20.452/135.168 (15%)	12.157/135.168 (8%)
Occupied slices	47.576/67.584 (70%)	43.602/67.584 (67%)
4 input LUTs	59.868/135.168 (44%)	53.417/135.168 (39%)
Bonded IOBs	41/768 (5%)	41/768 (5%)

Table 8: LAR timing results.

Transformation	Automatic	Manual
Development time	30%	100%
Maximum frequency (MHz)	61.43	85.27
Latency (ms)	0.42	0.12
Throughput frequency (MHz)	3.5	5.6
Processing time (ms/image)	35	19
Global image processing (FPS)	34	53

After the synthesis of the design, we applied a simulation stream of compressed videos. Table 8 below presents the timing results of a CIF (352×288) image size video.

For area consumption, the difference is not considerable for LUTs and occupied slices and it can be explained by the fact that the transformation applies a general modification whatever the complexity of the actor. Also, the fact of creating an internal buffer for every input port involves more area consumption.

Concerning the timing results, the automatic and the manual transformed designs performances remain close and acceptable. The latency difference is explained by the fact that the untagged actions, as always given priority over the rest of actions, promote the data reading. It means that, as long as there is data in the FIFO, the untagged action fires even if there are enough data to fire the processing actions. This problem will also be resolved by further optimizations of the buffer size.

7. Conclusion

This paper presented an automatic transformation of RVC-CAL from high- to low-level description. The purpose of this work is to find a general solution to automate the whole hardware generation flow from system level. This transformation allows avoiding structures that are not understandable by RVC-CAL hardware compilers. We applied this automatic transformation on the 29 actors of MPEG4 part2 video intradecoder and successfully obtained the same behavior of the multitoken design and a synthesizable hardware implementation. To change the test context, we automatically transformed a high-level design of the LAR still image codec and obtained relatively acceptable results.

Several optimization processes were added to the transformation to reduce the area consumption about 50%. The transformation process is currently generalized for all actors.

The most important in this work is that we contributed in making RVC-CAL hardware generation very rapid with an average gain of 75% of conception, development, and validation time compared with manual approach. We insured that the generation is applicable at system level whatever the complexity of the actor.

Currently, improvements are also in progress to customize the transformation depending on the actor complexity analysis. A future work will be the study of the impact of the transformation on the power consumption of the generated implementation.

Acknowledgments

Special thanks to Matthieu Wipliez, Damien De Saint-Jorre, and Hervé Yviquel for their relevant contributions in the source code.

References

[1] B. Bailey, G. Martin, and A. Piziali, *ESL Design and Verification: A Prescription for Electronic System-Level Methodology*, The Morgan Kaufmann Series in Systems on Silicon, Morgan Kaufmann, 2007.

[2] "Synopsys: Synphony C compiler," In ESL design and verification: a prescription for electronic system-level methodology, http://www.synopsys.com/systems/blockDesign/HLS/pages/SynphonyC-Compiler.aspx

[3] "Mentor Graphics: Designing High-Performance DSP Hardware Using Catapult C Synthesis and the Altera Accelerated Libraries," In ESL design and verification: a prescription for electronic system-level methodology, http://www.altera.com/literature/wp/wp-01039.pdf.

[4] "Mentor Graphics: Catapult C," In ESL design and verification: a prescription for electronic system-level methodology, 2010, http://www.mentor.com/esl/catapult/overview.

[5] F. Plavec, Z. Vranesic, and S. Brown, "Towards compilation of streaming programs into FPGA hardware," in *Proceedings of the Forum on Specification, Verification and Design Languages (FDL '08)*, pp. 67–72, September 2008.

[6] D. Lau, O. Pritchard, and P. Molson, "Automated generation of hardware accelerators with direct memory access from ANSI/ISO standard C functions," in *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '06)*, pp. 45–56, April 2006.

[7] T. M. Bhatt and D. McCain, "Matlab as a development environment for FPGA design," in *Proceedings of the 42nd annual Design Automation Conference (DAC '05)*, ACM, New York, NY, USA, 2005.

[8] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Design and Test of Computers*, vol. 26, no. 4, pp. 8–17, 2009.

[9] P. Coussy, C. Chavet, P. Bomel, D. Heller, E. Senn, and E. Martin, "GAUT: a high-level synthesis tool for DSP applications," in *High-Level Synthesis: From Algorithm to Digital Circuits*, P. C. A. Morawiec, Ed., Springer, 2008.

[10] J. Eker and J. Janneck, "CAL language report," ERL Technical Memo UCB/ERL M03/48, University of California at Berkeley, 2003.

[11] C. Brooks, E. Lee, X. Liu, S. Neuendorer, and Y. Zhao, Eds., "HZ: PtolemyII—heterogeneous concurrent modeling and design in Java (Volume 1: introduction to ptolemyII)," Technical Memorandum UCB/ERL M04/27, University of California, Berkeley, Calif, USA, 2004.

[12] E. A. Lee and T. M. Parks, "Dataow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, Article ID 773801, 1995.

[13] G. Kahn, in *Proceedings of the IFIP Congress The Semantics of a Simple Language for Parallel Programming. In Information Processing*, J. L. Rosenfeld, Ed., pp. 471–475, North-Holland, New York, NY, USA, 1974.

[14] M. Mattavelli, I. Amer, and M. Raulet, "The reconfigurable video coding standard," *IEEE Signal Processing Magazine*, vol. 27, no. 3, pp. 159–167, 2010.

[15] K. Jerbi, M. Wipliez, M. Raulet, M. Babel, O. Deforges, and M. Abid, "Automatic method for efficient hardware implementation from rvc-cal dataflow: a lar coder baseline case study," *Journal Of Convergence*, vol. 1, Article ID 8592, 2010.

[16] K. Jerbi, M. Raulet, O. Deforges, and M. Abid, "Automatic generation of synthesizable hardware implementation from high level RVC-CAL design," in *Proceedings of the 37th International Conference on Acoustics Speech and Signal Processing (ICASSP '12)*, pp. 1597–1600, 2012.

[17] J. Janneck, I. Miller, D. Parlour, G. Roquier, M. Wipliez, and M. Raulet, "Synthesizing hardware from dataflow programs: an mpeg-4 simple profile decoder case study," *Journal of Signal Processing Systems*, vol. 63, no. 2, pp. 241–249, 2009.

[18] M. Mattavelli, J. W. Janneck, and M. Raulet, "MPEG reconfigurable video coding," in *Handbook of Signal Processing Systems*, S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, Eds., pp. 43–67, Springer, 2010.

[19] J. W. Janneck, M. Mattavelli, M. Raulet, and M. Wipliez, "Reconfigurable video coding: a stream programming approach to the specification of new video coding standards," in *Proceedings of the ACM SIGMM Conference on Multimedia Systems (MMSys '10)*, pp. 223–234, New York, NY, USA, February 2010.

[20] S. Bhattacharyya, G. Brebner, J. Eker et al., "OpenDF—a dataflow toolset for reconfigurable hardware and multicore systems," in *1st Swedish Workshop on MultiCore Computing (MCC '08)*, Ronneby, Sweden, November 2008.

[21] R. Gu, J. W. Janneck, S. S. Bhattacharyya, M. Raulet, M. Wipliez, and W. Plishker, "Exploring the concurrency of an MPEG RVC decoder based on dataflow program analysis," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 19, no. 11, pp. 1646–1657, 2009.

[22] G. Roquier, M. Wipliez, M. Raulet, J. W. Janneck, I. D. Miller, and D. B. Parlour, "Automatic software synthesis of dataflow program: an MPEG-4 simplpe profile decoder case study," in *Proceedings of IEEE Workshop on Signal Processing Systems (SiPS '08)*, pp. 281–286, Washington, DC, USA, October 2008.

[23] M. Wipliez, G. Roquier, and J. F. Nezan, "Software code generation for the RVC-CAL language," *Journal of Signal Processing Systems*, vol. 63, no. 2, pp. 203–213, 2011.

[24] J. Eker and J. W. Janneck, "A structured description of dataflow actors and its application," Technical Memorandum UCB/ERL M03/13, Electronics Research Laboratory, University of California at Berkeley, 2003.

[25] R. K. Megalingam, K. B. Venkat, S. V. Vineeth, M. Mithun, and R. Srikumar, "Hardware implementation of low power, high speed DCT/IDCT based digital image watermarking," in *Proceedings of the International Conference on Computer Technology and Development (ICCTD '09)*, pp. 535–539, November 2009.

[26] O. Déforges, M. Babel, L. Bédat, and J. Ronsin, "Color LAR codec: a color image representation and compression scheme based on local resolution adjustment and self-extracting

region representation," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 17, no. 8, pp. 974–987, 2007.

[27] O. Deforges and M. Babel, "LAR method: from algorithm to synthesis for an embedded low complexity image coder," in *Inproceedings of the 3rd International Design and Test Workshop (IDT '08)*, pp. 187–192, December 2008.

[28] J. Poncin, "Utilisation de la transformation de Hadamard pour le codage et la compression de signaux d'images," *Annales des Télécommunications*, vol. 26, no. 7-8, pp. 235–252, 1971.

[29] R. F. Rice, "Some practical universal noiseless coding techniques," Technical Report 79–22, 1979.

[30] S. W. Golomb, "Run length codings," *IEEE Transactions on Information Theory*, vol. 12, no. 7, Article ID 399401, 1966.

[31] C. H. Chang, M. H. Chang, and W. Hwang, "A flexible two-layer external memory management for H.264/AVC decoder," in *Inproceedings of the 20th Anniversary IEEE International SOC Conference*, pp. 219–222, September 2007.