AUTOMATIC GENERATION OF SYNTHESIZABLE HARDWARE IMPLEMENTATION FROM HIGH LEVEL RVC-CAL DESCRIPTION

Khaled Jerbi ⋆† Mickey Raulet † Olivier Déforges † Mohamed Abid *

†IETR/INSA, UMR CNRS 6164, F-35043 Rennes
Email: FirstName.Name@insa-rennes.fr
⋆CES Lab. National Engineering School of Sfax
Email: FirstName.Name@enis.rnu.tn

ABSTRACT
Data process algorithms are increasing in complexity especially for image and video coding. Therefore, hardware development using directly hardware description languages (HDL) such as VHDL or Verilog is a difficult task. Current research axes in this context are introducing new methodologies to automate the generation of such descriptions. In our work we adopted a high level and target-independent language called CAL (Caltech Actor Language). This language is associated with a set of tools to easily design dataflow applications and also a hardware compiler to automatically generate the implementation. Before the modifications presented in this paper, the existing CAL hardware back-end did not support some high-level features of the CAL language. Consequently, high-level designed actors had to be manually transformed to be synthesizable. In this paper, we introduce a general automatic transformation of CAL descriptions to make these structures compliant and synthesizable. This transformation analyses the CAL code, detects the target features and makes the required changes to obtain synthesizable code while keeping the same application behavior. This work resolves the main bottleneck of the hardware generation flow from CAL designs.

Index Terms— RVC-CAL, Data flow computing, CAL compiler, Automatic transformation

1. INTRODUCTION
Watching and exchanging videos on the Internet or mobile gadgets by billions of people all over the world is the current phenomenon. Moreover, user requirements of high quality are also growing which has caused a noteworthy increase in the complexity of the algorithms of video codecs. To be used, these algorithms have to be implemented on a target architecture that specifies the description language depending on its nature: hardware or software. In this context, CAL Actor Language [1] was introduced in the Ptolemy II project [2] as a dataflow target agnostic language. The MPEG community standardized the RVC-CAL language in the MPEG-RVC (Reconfigurable Video Coding) standard [3]. 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. Xilinx developed a hardware compiler called OpenForge1 [4] that generates hardware implementation using an Intermediate Representation (IR) called XLIM. The bottleneck of hardware generation using OpenForge is the fact that RVC-CAL code generally contains high level structures that are not compatible with this hardware compiler.

On the other hand, an RVC-CAL compiler called Orcc2 [5] is in development as a tool that compiles a network of actors and encompasses several backends that generate code for different targets: C, LLVM, Java and notably XLIM. We notice that an Orcc VHDL backend [6] is under development but it is not enough mature to compile loop nodes. Therefore, we proposed to use Openforge and the IR of Orcc to add automatic transformations making any RVC-CAL design synthesizable.

This paper presents an automatic transformation of RVC-CAL code to modify the unsupported structures in the IR of Orcc. The transformation tool analyzes the RVC-CAL code and achieves the required transformations to obtain synthesizable code whatever the complexity of the considered actor. In section 2, we explain the main notions of the RVC-CAL language and its behavioral structures and mechanisms. The proposed transformation process is detailed in section 3 and finally hardware implementation results of MPEG4 Part2 decoder case study are presented in section 4.

2. BACKGROUND
The execution of an RVC-CAL code is based on the exchange of data tokens between computational entities called 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

1http://openforge.sf.net
2http://orcc.sf.net
may change the state of the actor in case of an FSM. An RVC-CAL dataflow model is shown in the network of Figure 1.

\[ \text{Fig. 1. CAL actor model} \]

The actor execution, so called firing, is based on the Dataflow Process Network (DPN) principle \([7]\) derived from the Kahn Process Network (KPN) \([8]\). Let \( \Omega \) be the universe of all tokens values exchanged by the actors and \( \mathbb{S} = \Omega^* \) the set of all finite sequences in \( \Omega \). We denote the length of a sequence \( s \in \mathbb{S}^k \) by \( |s| \) and the empty sequence by \( \lambda \). 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, \( s_0 = [\lambda, [t_0], [t_2]] \) 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 \( |s_1| = [1,1] \). A firing rule is called multi-token \( iff \) \( \exists e \in |s| : e > 1 \) otherwise it is called a mono-token rule. The limitation of Openforge is the fact that it does not support multi-token rules which are omnipresent in most actors. Our work consists of automatically transforming the data consumption from multi-token to mono-token while preserving the same actor behavior: firing rules and transitions are detailed below.

2.1. Actor firing

A dataflow actor is defined with a pair \( < f, R > \) such as:
- \( f : \mathbb{S}^m \rightarrow \mathbb{S}^n \) is the firing function
- \( R \subseteq \mathbb{S}^m \) are the firing rules
- 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:

\[
\begin{align*}
\{ g_{j,k} : [x]x > 0 \} \\
\{ r_j = [t_0 \in g_{j,k}, [t_1, t_2, t_3]] \}
\end{align*}
\]

which 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 \( iff \)

- There are enough tokens in the input FIFO
- A guard condition returns true

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

2.2. Actor transition

The FSM transition system of an actor is defined with \( < \sigma_0, \Sigma, \tau, \prec > \) where \( \Sigma \) is the set of all the states of the actor, \( \sigma_0 \) is the initial state, \( \prec \) is a priority relation and \( \tau \subseteq \Sigma \times \Sigma \times \Sigma \times \Sigma \) is the set of all possible transitions. A transition from a state \( \sigma \) to a state \( \sigma' \) with a consumption of sequence \( s \in \mathbb{S}^m \) and a produced sequence \( s' \in \mathbb{S}^n \) is defined with \( (\sigma, s, s', \sigma') \) and denoted:

\[ \sigma \xrightarrow{\tau} s' \prec \sigma' \]

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 \([9]\) a transition \( \sigma \xrightarrow{\tau} s' \prec \sigma' \) is enabled \( iff \):

\[ \neg \exists \sigma \xrightarrow{\tau} s'' \ni p \in \Sigma \wedge \sigma \xrightarrow{\tau} s'' \succ \sigma \xrightarrow{\tau} s' \]

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 macro-behavior of the actor.

3. THE PROPOSED METHOD

As shown in Figure 2, our transformation acts on the IR representation of the front-end. The HDL implementation is later generated using the Xlim back-end of Orcc followed by OpenForge.

\[ \text{Fig. 2. Automatic transformation localization in Orcc compiling process} \]
3.1. Actor transformation principle

Let us consider an actor with a multi-token firing rule \( r \in S^k \) such as \(|r| = [r_0, r_1, \ldots, r_{k-1}]\), this rule fires a multi-token action \( a \) realizing the transition \( \frac{a}{\text{read}} \rightarrow \text{target} \) and \( I \) 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:

\[
\forall i \in I, \exists \rho \in \mathbb{R} : \|r\| = 1
\]

with \( \rho \) a mono-token firing rule of an untagged action \( \text{untagged}_i \), \( g_\rho \) is the guard of \( \rho \) 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 multi-token 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 non-determinism 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 action realizing the FSM passage \( \frac{\text{source}}{\text{transition}} \rightarrow \text{read} \) and has the same priority order of the deleted multi-token action but has no process. The read actions loops in the read state with the transition \( t = \text{read} \frac{\text{read}}{\text{readDone}} \rightarrow \text{read} \). Then the loop is exited when all necessary tokens are read using a read done action and a transition to the process state \( t' = \text{read} \frac{\text{readDone}}{\text{process}} \rightarrow \text{process} \). The treatment of the multi-token action is put in a process action with a transition \( \frac{\text{process}}{\text{write}} \rightarrow \text{write} \). The multi-token outputs are also transformed into a writing loop with write actions that store data directly in the output FIFO associated with a transition \( w = \text{write} \frac{\text{write}}{\text{writeDone}} \rightarrow \text{write} \) and a write done action that insures the FSM transition \( w' = \text{write} \frac{\text{writeDone}}{\text{target}} \rightarrow w \).

For example, the actor \( A \) of Figure 3 is defined with \( f : S^3 \rightarrow S^2 \) with a multi-token firing rule:

\[
r \in S^3 : r = [t_0, t_1, t_2, t_3, t_4, t_5].
\]

actor A () int IN1, int IN2, int IN3 ==> int OUT1, int OUT2:
  a; action
  do
    (treatment)
  end
end

Fig. 3. RVC-CAL code of actor A

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 macro-block of Figure 4.

3.2. FSM creation cases

We consider an example of an actor defined as \( f : S^3 \rightarrow S^2 \) containing the actions \( a_1 \ldots a_5 \) such as \( a_3 \) is the only action applying a multi-token firing rule \( r \in S^3 \). Creating an FSM only for action \( a_3 \) is not appropriate because \( a_1, a_2, a_4, a_5 \) 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 macro-block of \( a_3 \) (previously presented in Figure 4). The resulting FSM is presented in Figure 5. We now suppose the same actor scheduled with an initial FSM as shown in Figure 6. The transition \( t = S1 \frac{a_3}{\text{proc}} \rightarrow S2 \) is substituted with the macro-block of \( a_3 \) as shown in Figure 7.

Fig. 4. Created FSM macro-block

The achieved automatic transformation was applied on MPEG4 Part2 intra decoder (see design in Orcc Applications3) which

\[3\text{http://orc-apps.sf.net}\]

4. RESULTS
contains 29 actors. The HDL generated code was implemented on a virtex4 (xc4vlx160-12ff1148) and the area consumption results we obtained are presented in table1. After the synthesis of the design, we applied a simulation stream of compressed videos. Table 2 below presents the timing results of a QCIF (176x144) image size video.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Slice Flip Flops</td>
<td>21,624/135,168 (15%)</td>
<td></td>
</tr>
<tr>
<td>Occupied Slices</td>
<td>45,574/67,584 (67%)</td>
<td></td>
</tr>
<tr>
<td>4 input LUTs</td>
<td>68,962/135,168 (51%)</td>
<td></td>
</tr>
<tr>
<td>FIFO16/RAMB16s</td>
<td>14/288 (4%)</td>
<td></td>
</tr>
<tr>
<td>Bonded IOBs</td>
<td>107/768 (13%)</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Design area consumption

<table>
<thead>
<tr>
<th>Image size</th>
<th>transformed design</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum frequency(MHz)</td>
<td>26.4</td>
</tr>
<tr>
<td>Latency(μs)</td>
<td>381.8</td>
</tr>
<tr>
<td>Cadency(MHz)</td>
<td>1.9</td>
</tr>
<tr>
<td>Processing time(ms/image)</td>
<td>13.55</td>
</tr>
<tr>
<td>Global image throughput (FPS)</td>
<td>73.8</td>
</tr>
</tbody>
</table>

Table 2. MPEG4 decoder timing results

5. CONCLUSION

This paper presented an automatic transformation of RVC-CAL from high to low level description. 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 intra decoder and successfully obtained the same behavior of the multi-token design and a synthesizable hardware implementation. Currently, we are optimizing the transformation to reduce the memory consumption. The transformation process is generalized for all actors. Therefore, 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.

6. ACKNOWLEDGMENTS

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

7. REFERENCES


