Automatic Method For Efficient Hardware Implementation From RVC-CAL Dataflow: A LAR Coder baseline Case Study

Khaled Jerbi, Matthieu Wipliez, Mickaël Raulet,
Marie Babel, Olivier Déforges
INSA of Rennes UMR CNRS 6164
IETR Group
Rennes, France
khaled.jerbi@insa-rennes.fr

Mohamed Abid
National school of engineers of Sfax
CES Lab
Sfax, Tunisia
mohamed.abid@enis.rnu.tn

***Abstract*— Implementing an algorithm into hardware platforms is generally not an easy task. The algorithm, typically described in a high-level specification language, must be translated into a low-level HDL language. The difference between models of computation (sequential versus fine-grained parallel) limits the efficiency of automatic translation. On the other hand, manual implementation is time-consuming, because the designer must take care of low-level details, and write test benches to test the implementation's behaviour. This paper presents a global design method, from high level description to implementation. The first step consists of the description of an algorithm as a dataflow program using RVC-CAL language. The next step is the functional verification of this description using a software framework. The final step consists of an automatic generation of an efficient hardware implementation from the dataflow program. The objective is to spend the greater part of the conception time in an open source software platform. We use this method to quickly prototype and generate hardware implementation of a baseline part of the LAR coder, from an RVC-CAL description.**

Keywords- Dataflow programming, RVC CAL language, LAR coder, fast hardware implementation, Orcc

I. INTRODUCTION

The complexity of signal processing algorithms is continually increasing, which involves a very long description code. For designers this code is very hard to implement on hardware platforms. Hardware implementation requires the description of the process using an HDL language like VHDL or Verilog. These dataflow designs are not easy to develop and especially not to validate. The validation of a dataflow design requires the development of stimulus codes, such as a VHDL test bench, and the use of simulation tools. This explains the gap between validating and implementing a process. Therefore designers can hardly satisfy the time-to-market constraints. To solve this problem, designers tend to design solutions to describe the process in at a higher level. In the video coding field, a new high level description language for dataflow applications, called RVC-CAL (Reconfigurable Video Coding) [1], was standardised by the MPEG community through the MPEG-RVC standard [2]. This standard provides a framework to define different codecs by combining communicating blocks developed in RVC-CAL.

The objective of our work is to provide a hardware implementation generated from a high level description using the RVC-CAL programming language [3]. We use an efficient hardware generator from RVC-CAL called Cal2HDL [4], [5]. It uses an intermediate representation of the OpenDF project [6]. Cal2HDL supports all the structures of the RVC-CAL language except the multi token and some loop structures. Consequently, we have to manually change these structures to their equivalents supported by the tool, which involves a new code and thus an additional verification step.

In this paper, we introduce an original global approach to expedite the validation of an RVC-CAL design and consequently the dataflow generation. This approach was applied to the LAR (Locally Adaptive Resolution) image coder [7] to provide a realistic application context. The current design does not contain the full LAR coder, but we already achieved some of the main parts of the LAR baseline with an RVC-CAL description.

In Section II, we present the method and the languages and frameworks used. In Section III, the LAR coding principle is detailed. Finally Section IV shows an application of the method on the LAR coder baseline and also provides some implementation results.

II. DATAFLOW PROGRAMMING FOR HARDWARE IMPLEMENTATION

The purpose of this work is to more rapidly obtain a dataflow description from an RVC-CAL design. In the following, we present a new global method for the functional verification of an RVC-CAL code. As presented in Figure 1, the design is described at a high level with RVC-CAL language. Then a software platform is used for functional validation and FIFO sizing. Once the code is correct, it undergoes a modification to be synthesisable with Cal2HDL by unrolling the loops and the repeat structures. The validation of this code is realised with the same software platform. The implementation is finally insured by using a hardware synthesis and prototyping platform.

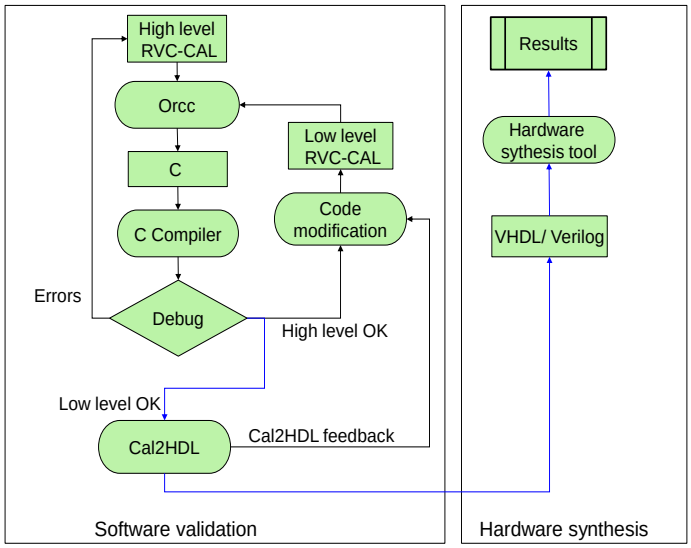

Figure 1. Method overview

A. *Dataflow programming with RVC-CAL language*

MPEG RVC is under development as part of the MPEG-B standard [3], which defines the framework and the language used to describe components. RVC-CAL [3] is a textual and domain specific language for writing dataflow models (Figure 2), more precisely for defining *actors* of a dataflow model in a high level description. An actor represents an autonomous entity and a composition of actors explicitly describes the concurrency of an application. The RVC-CAL Actor Language has been defined to be platform independent and retargetable to a rich variety of platforms.

An RVC-CAL *actor* is a computational entity with input ports, output ports, states and parameters. An *actor* communicates with other actors by sending and receiving *tokens* (atomic pieces of data) through its ports. An actor can contain several *actions*. An *action* defines a computation, which consumes sequences of tokens from input ports and produces sequences of tokens to output ports. Actions have data-dependent conditions for their execution. The execution of an action may change the actor's internal state, so that the produced output sequences are functions of the consumed input sequences and of the current actor state. RVC-CAL supports higher-level constructs such as multiple-token reads/writes and list generators.

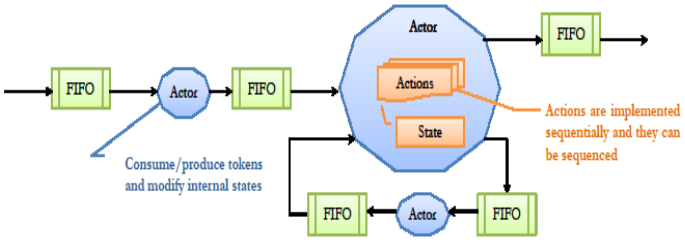

Figure 2. CAL dataflow model

.

B. *Functional verification on a software platform*

CAL code validation is usually based on the OpenDF simulator. It has to be stimulated with manually given tokens via data generation and data display actors. The result is a set of values that have to be verified. The originality of our approach is to realise the CAL validation step using an Open RVC-CAL Compiler (Orcc) [8]. Orcc is an open source software (http://sourceforge.net/projects/Orcc/), developed at the IETR laboratory of the INSA of Rennes. Orcc is a source-to-source compiler that compiles RVC-CAL dataflow programs to a target language. It is composed of 3 main parts:

1- The front-end that parses the code into an Abstract syntax tree.

2- The middle-end which analyses the behaviour of actors.

3- The back-end that generates the target language through predefines templates. Available target languages are C, C++, Xlim, Promela, Java, LLVM and VHDL.

In our work, we use the C backend of Orcc. This choice is explained by the fact that C language is the most used language in software programming. After compilation, we can easily assign a video or an image as an input and visualise the output. It is very important to mention that Orcc compilation, video processing and display using the C compiler, are very fast steps. In addition, the software debug is less time consuming than hardware one. Consequently, the CAL errors are more easily detected and corrected faster. Moreover, we can use Orcc to define the optimal FIFO sizes for lower memory consumption in the hardware implementation.

C. *HDL generation*

We used Cal2HDL for dataflow generation. This tool parses the CAL code, generates an XML representation for each actor and synthesises the static single assignment (SSA) threads into circuits based on basic operators. The final description is made up of a Verilog file for each actor and a VHDL file for the top: the highest hierarchical representation of the design connections. The connection between the actors is insured by synchronous or asynchronous FIFO buffers.

Currently, Cal2HDL does not support all the structures used in a RVC-CAL description such as repeats and loops. These structures have to be manually modified into several actions managed by a finite state machine. Figure 3 shows an example of an action writing the 16 values of a buffer named "tab" in the output port called "OUT". The instruction "repeat 16" enables access to the first 16 values of the buffer "tab".

```
write: action ==> OUT:[tab] repeat 16
end
```

Figure 3. High level RVC-CAL example

This action has to be modified in the code presented in Figure 4. The modifications consist of deleting the "repeat" structure to have an action that only produces one token and repeats the basic action 16 times. The repetition process starts by executing the "write" action until the "write_done" action is validated. Everything has to be managed by a finite state machine defined by the structure "schedule fsm" in Figure 4.

```
write: action ==> OUT:[out]
do
  out := tab[counter];
  counter := counter + 1;
end

write_done: action ==>
guard
  counter = 16
do
        counter := 0;
end

schedule fsm write:
  write (write     ) --> write;
  write (write_done) --> nextstate;
...
end
```

Figure 4. Low level RVC-CAL example

After this transformation we obtain a synthesizable code and Cal2HDL can generate the adequate hardware description.

III. THE LAR CODER

To apply and validate the global method introduced, we chose the LAR image coder. This coder is developed at the IETR/ INSA of Rennes laboratory. It is based on the idea that spatial coding can be locally dependent on the activity in the image. Thus, the higher the activity the lower is the resolution. This activity is dependent from the variation or the uniformity of the local luminance, which can be detected using a morphological gradient that will be further explained. Another aspect of LAR coding is based on considering the fact 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 modelled by (1) where I is the original image, I' is the global information image and (I-I') is the error image. The dynamic range of the error image is consequently dependent on the local activity. In uniform regions, I' values are close or equal to I consequently (I-I') values are around zero with a low dynamic range.

$$I = I' + (I-I') \qquad (1)$$

Considering these principles, the LAR coder concept (Figure 5) is composed of two parts: the FLAT LAR [9] 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 different types of application. In this paper, we focus on the baseline coder. Its mechanisms are detailed in this section

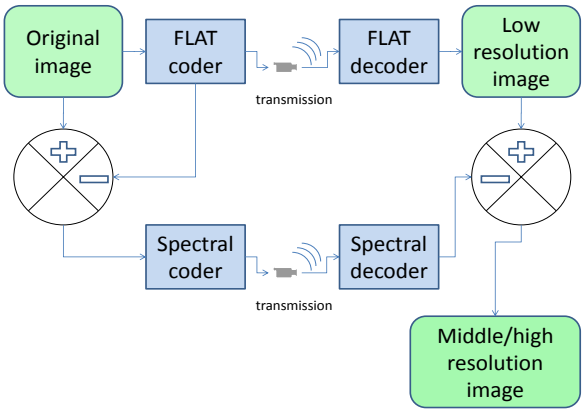

Figure 5. LAR concept

A. 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).

1) Partitioning: In this part, a Quad-Tree partitioning is applied to the image pixels. The principle is to consider the lowest block size (2x2) 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 (Nx2) x (Nx2) size block is required. This process is recursively applied to the whole image blocks. The overall output is the block size image.

2) 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 placed on the block, as presented in the example shown in Figure 6.

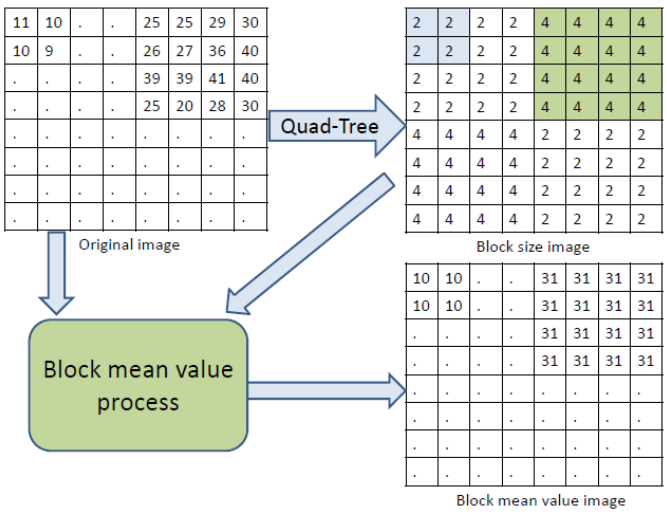

Figure 6. Block mean value process example

3) The DPCM: The DPCM process is based on the prediction of neighbour values and the quantisation of the block mean value image. The observation that a pixel value is mostly equal to a neighbouring one led to the following

estimation algorithm. If we consider the pixels in Figure 7, X value is estimated with the algorithm: If |B-C| < |A-B| then X = A else X = C

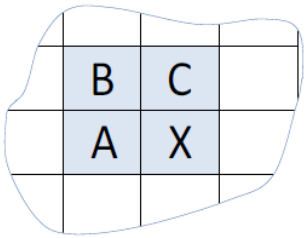

Figure 7. DPCM prediction of neighbouring pixels

B. Spectral coder: the Hadamard transform

The spectral coder, also called the texture coder, is composed of a variable block size Hadamard transform [10] and the Golomb-Rice [11] [12] entropy coder.

The Hadamard transform derives from a generalised class of the Fourier transform. It consists of a multiplication of a 2^m x2^m matrix by an Hadamard matrix (H_m) of the same size. The transform is defined as follows:

H_0 is the identity matrix so H_0=1. For any m>0, H_m is then deducted recursively by (2).

$$H_m = \frac{1}{\sqrt{2}} \begin{vmatrix} H_{m-1} & H_{m-1} \\ H_{m-1} & -H_{m-1} \end{vmatrix} \quad (2)$$

Here are examples of Hadamard matrices:

$$H_0 = 1 \ ,$$

$$H_1 = \frac{1}{\sqrt{2}} \begin{vmatrix} 1 & 1 \\ 1 & -1 \end{vmatrix} \ ,$$

$$H_2 = \frac{1}{\sqrt{2}} \begin{vmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{vmatrix} \ , \text{ etc ...}$$

IV. HARDWARE IMPLEMENTATION OF THE LAR CODER BASELINE

We have already developed and implemented a subset of the Flat LAR coder using RVC-CAL in previous work as described in [13]. Therefore, from this preliminary implementation we would almost achieve the implementation of the whole LAR codec following MPEG-RVC standard recommendations.

This Section explains the mechanisms of the Hadamard transform and the Quad-Tree used in the implementation. Dataflow implementation and synthesis results are also presented and discussed.

A. Hardware implementation

The LAR coding is dependent on the content of the image. It applies to the Quad-Tree a morphological gradient to extract information about local activity on the image. The output is the block size image represented by variable size blocks: 2x2, 4x4 or 8x8. Using the block size image, the Hadamard transform applies the adequate transform onto the corresponding block. It means that if we have a block size of 2X2 in the size image this block will undergo a 2X2 Hadamard (H_1) and normalisation specific to the 2X2 blocks. This process is identically applied for 4X4 and 8X8 blocks. A quantisation step, adapted to the current block size, is applied on the Hadamard output image. For each block size, a quantisation matrix is pre-defined. In practice the normalisation during the Hadamard transform is postponed to be achieved with the quantisation step so to decrease the noise due to successive divisions. The implemented LAR is presented in Figure 8.

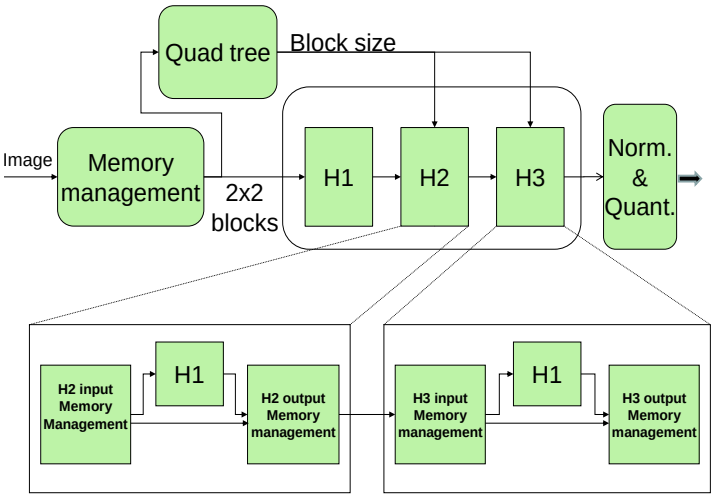

Figure 8. LAR baseline developed model

As a first step, the memory management block stores the pixel values of the original image line by line. Once an 8x8 block is obtained, the actor divides it into sixteen 2x2 blocks and sends them in a specific order as presented in Figure 10.

This order is very important to improve the performance of the remaining actors. In fact, considering Figure 10, when the tokens are so ordered the first 4 tokens correspond to the first 2x2 block, the first 16 tokens to the first 4x4 block, etc. Consequently, and as presented in Figure 8, the output of H1 is automatically the input of H2 and the output of H2 is automatically the input of H3.

In the Quad-Tree, this order is also crucial. As presented in Figure 9, the superposition of the same actor (max for example) three times provides in the output of the first actor the maximum values of 2x2 blocks, in the output of the second actor the maximum values of 4x4 blocks and finally the maximum values of 8x8 blocks in the output of the third one. Using the maximum and minimum values the morphological gradient in the Gradstep actors can be processed to extract the block size image. The same idea 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

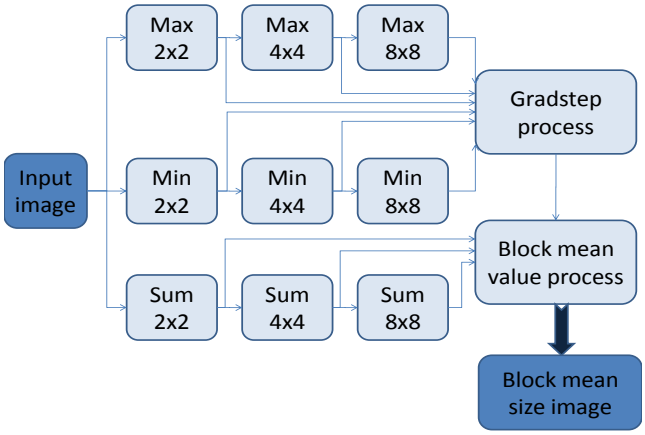

Figure 9. Quad-Tree design

We also notice that an (H_2) transform can be achieved using the (H_1) results of the four 2X2 blocks constituting the 4X4 block. The same observation can be made for (H_3). This ascertainment is very important in order to decrease the complexity of the process. In fact, the Hadamard transform of the LAR applies an (H_1) transform for the whole image, then it applies the (H_2) transform only for the 4X4 and 8X8 blocks, and the (H_3) transform only for the 8X8 blocks. (H_2) and (H_3) transforms are different from the full transforms as they are much less complex. Consequently, as shown in Figure 8, we designed the H2 and the H3 using H1 actors associated with memory management units. They sort tokens into an adequate order and, considering the block size, determine whether the block is going to undergo the transform or not.

It is very important to mention that almost all actors have been developed with generic variables for memory sizes or gradsteps, which means that the design are flexible for easy transformation from one image size to another, or for adding higher Hadamard processes (H4, H5 ...).

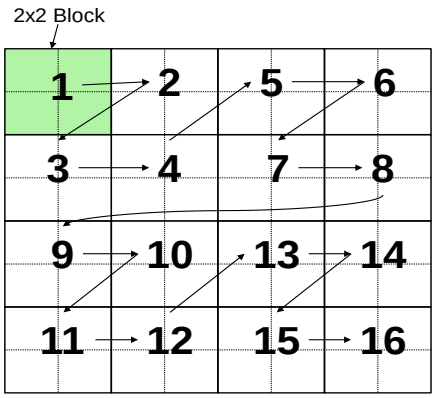

Figure 10. Memory management unit output order

The different actors of the LAR baseline coder have first been developed with a high level RVC-CAL description for a 352x288 image size. To optimise the transform, we added a ping-pong data management algorithm. The principle of this algorithm is to avoid the repetitive latency caused by data storage while reading tokens, by combining the tokens' reading and writing in the same action. The idea is to write the input data in half of a memory size and then to use this data while writing in the other half. Finally we just have to switch the reading and the writing pointers to the opposite parts of the memory. An example of ping-pong memory management of a 4 buffers memory is presented in Figure 1.

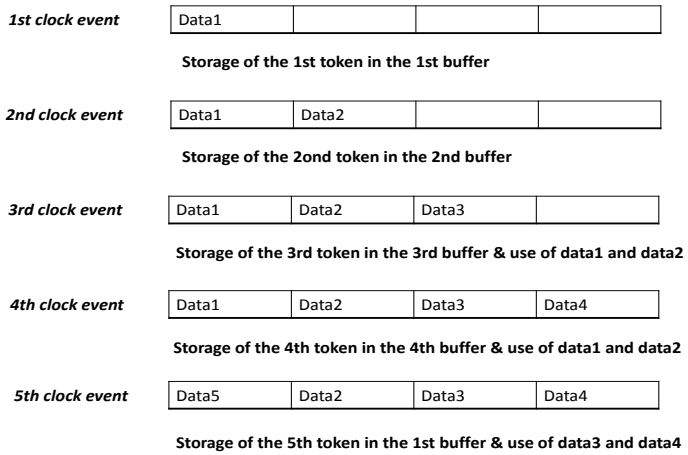

Figure 11. Ping pong example of a 4-buffer size memory management

In RVC-CAL language, the solution is to use pointer functions, as *ra()* and *wa()* in the example in figure Figure 12. After reading and filling the half of the memory, a boolean flag *half* changes the read and the write pointers represented by ra() and wa(). Thus, an alternation of read and write is created in the action *gradient*. Consequently, while writing tokens, that actor is reading new ones for the next process which considerably decreases the processing time.

Timing performances have then increased considerably. Other optimisations can be added by treatment anticipation, but they have not been added in this case because the design would be a low level one.

A reverse Hadamard block was added for validation. The whole design was compiled with Orcc to obtain the C code of the actors. C code was compiled with a C compiler. To test the design we applied images and videos in the inputs. The objective was to obtain an output exactly equal to the input, as presented in Figure 1.

```
      bool half := false;
function wa() --> int :
  bitor( bitand(cnt,BLK_SZ*BLK_SZ-1),
  if half then BLK_SZ*BLK_SZ else 0 end )
end

function ra() --> int :
  bitor( bitand(cnt,BLK_SZ*BLK_SZ-1),
  if half then 0 else BLK_SZ*BLK_SZ end )
end

read_bloc_size : action BLK_SZ_IN:[size_in] -->
do
 sz_in[cnt]:=size_in;
 cnt:=cnt+1;
end

done : action MAX:[max], MIN:[min] -->
guard cnt= BLK_SZ*BLK_SZ
do
max_tmp:=max;
min_tmp:=min;
half := not half;
cnt:=0;
end

gradient : action BLK_SZ_IN:[size_in] --> BLK_SZ_OUT:[out]
var
  int out
do
  sz_in[wa()] := size_in;
      if (max_tmp - min_tmp<GRADSTEP) then
              out:=BLK_SZ;
      else
              out:=sz_in[ra()];
      end
      cnt:=cnt+1;
end

schedule fsm read :
 read(read_bloc_size)  --> read;
 read(done)                  --> read_write;
 read_write(gradient)  --> read_write;
 read_write(done)            --> read_write;
end

priority
 done > read_bloc_size;
 done > gradient;
end
```

Figure 12. Ping-Pong memory management example

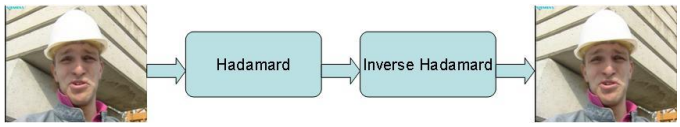

Figure 13. Software validation

Once the required pixel values are obtained the design is validated and consequently so is the RVC-CAL code. At this level, the VHDL/Verilog generation is not possible since Cal2HDL cannot generate code from the high level RVC-CAL. It was necessary to change the RVC-CAL code into another low level code synthesisable with Cal2HDL, as explained in Section II. Figure 14 shows the example of a "max 2x2" actor in a high level description with "repeat" and the "foreach" loops. This actor is translated to a low level one as presented in Figure 15.

```
actor max2x2()  uint(size=8) IN ==> uint(size=8) OUT:
  max2x2: action IN:[input] repeat 4 ==> OUT:[out]
  var
    int out:= 0
  do
    foreach int i in Integers(0, 3) do
      out:= if input[i]>out then input[i] else out end;
    end
  end

end
```

Figure 14. High level RVC-CAL example

```
actor max2x2()  int(size=9) IN ==> int(size=9) OUT:

 int(size=5) cpt:= 0;
 int(size=9) max:= 0;

 init: action IN: [in0] ==>
 do
  max:= in0 ;
 end

 compare : action IN: [in0]  ==>
 do
  if max < in0 then
   max:=in0;
  end
  cpt:= cpt+1;
 end

 send : action ==> OUT: [ max ]
 guard cpt = 3
 do
  cpt := 0 ;
 end

 schedule fsm init :
   init    ( init    ) --> compare;
   compare ( compare ) --> compare;
   compare ( send    ) --> init;
 end

 priority
   send>compare ;
 end

end
```

Figure 15. Low level RVC-CAL example

Thus, we obtained a dataflow implementation of the LAR baseline.

With our approach, the OpenDF validation of the classic method was avoided. In that classic method, we used to develop the RVC-CAL code and add actors for data generation and display: The actor of data generation is composed of a table containing the input image pixel values and some actions to consecutively put these values in the input port of the design. The design output data can be displayed using the "println()" function. Validation is consequently a tough and relentless value comparison process. The use of C compilers allows us to use images and videos for the test and we are able to have more information about an error when we have both the data values and image display.

B. Results

The HDL project manager environment used is Xilinx ISE Foundation 11.1 and the hardware simulation tool is ISE simulator (Full version). We manually developed the test bench by initialising the different signals and generating the stimulus values for the inputs.

After compilation, simulation, RTL synthesis and place and route on an FPGA: family=virtex4; device=xc4vsx35; Package=FF668; speed = -12, we obtain the area consumption results presented in Table I.

TABLE I. AREA CONSUMPTION FOR 352X288 IMAGE SIZE

Criterion	value
Slice Flip Flops	1.437/30.720 (4%)
Occupied Slices	2.027/15.360 (13%)
4 input LUTs	3.637/30.720 (11%)
FIFO16/RAMB16s	26/3192 (13%)
Bonded IOBs	99/448 (22%)

The time synthesis performances are mentioned in Table II

TABLE II. TIMING RESULTS

Criterion	352x288
Output frequency(MHz)	22.4
maximum frequency(MHz)	81.4
Latency(µs)	132.79
processing time(ms)	5.8
Minimum input arrival time before clock(ns)	12.134
Maximum output required time after clock(ns)	8.188
Maximum combinational path delay(ns)	5.083

Optimisation solutions are in development to decrease the latency and consequently increase the frequency. In terms of development time, the whole design took about 70 days to be achieved. It is very important to mention that over 90% of the conception time was achieved in the open source software platform where the debug and validation are easier and faster.

The most complex and time consuming part of the flow was the manual transformation of the RVC-CAL from high to low level. This can be explained by the fact that the code is longer and consequently harder to debug because of the inaccurate feedback of Cal2HDL. We are currently looking for solutions to automate this step. This task may be achieved by improving Cal2HDL Java source code or by using an intermediate representation of Orcc. The second case seems to be more feasible. However, this global framework introducing a software functional checking before synthesis process is significantly faster than a hardware implementation directly from the RVC-CAL description.

V. CONCLUSION

This paper presented a method to automatically generate an efficient functional hardware implementation from an RVC-CAL dataflow program. The method presented was used to obtain a hardware implementation of a LAR coder baseline. This transform implementation is a part of our work to achieve the implementation of the whole LAR image codec. We believe that frequency can be increased, and latency decreased, by further optimisation of memory management actors.

With our method, the design cycle of hardware implementation consists in carrying out the functional verification in a software environment, and testing the hardware implementation once the program is correct. The validation in the software platform is very fast and allows for huge data testing, notably images and videos with visualisation rather than using hardware simulators at every design step. We used Orcc to generate C code from RVC-CAL descriptions and to fix the optimal FIFO sizes. The C code was then compiled and run to test the program behaviour. The hardware implementation was obtained by automatically transforming the RVC-CAL descriptions with Cal2HDL. Cal2HDL generates a valuable but low level and hardly-understandable Verilog code. Concurrently, as presented recently in [14], Nicolas Siret et al. are using the IR of Orcc to generate a high level VHDL backend. Code generation with this backend is clearly faster than Cal2HDL, especially for high complexity designs, but it has not been sufficiently tested. Later this tool would be an alternative to Cal2HDL. Currently, high-level RVC-CAL descriptions must be manually transformed to lower-level code for Cal2HDL to be able to synthesise it. Automating this transformation will considerably reduce design time. Indeed, we are already starting working on this automatic transformation of the IR of Orcc.

REFERENCES

[1] J. Eker and J. Janneck, "CAL Language Report," Tech. Rep. ERL Technical Memo UCB/ERL M03/48, University of California at Berkeley, Dec. 2003.

[2] 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, 2009, dOI:10.1007/s11265-009-0399-3.

[3] ISO/IEC FDIS 23001-4: 2009, "Information Technology - MPEG systems technologies - Part 4: Codec Configuration Representation," 2009.

[4] 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.

[5] "Cal2HDL-openforge source" Available from:: http://openforge.sourceforge.net. [Accessed: December 2010]

[6] S. Bhattacharyya, G. Brebner, J. Eker, J. Janneck, M. Mattavelli, C. von Platen, and M. Raulet, "OpenDF - A Dataflow Toolset for Reconfigurable Hardware and Multicore Systems," First Swedish Workshop on Multi-Core Computing, MCC , Ronneby, Sweden, November 27-28, 2008.

[7] 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 Trans. Circuits Syst. Video Techn., vol. 17, no. 8, pp. 974–987, 2007.

[8] 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 MMSys '10: Proceedings of the first annual ACM SIGMM conference on Multimedia systems. New York, USA: ACM, pp. 223–234, 2010.

[9] O. Deforges and M. Babek, "Lar method: from algorithm to synthesis for an embedded low complexity image coder," IEEE 3rd International Design and Test Workshop, 2008.

[10] J. Poncin, "Utilisation de la transformation de hadamard pour le codage et la compression de signaux d'images," in Springer-Annals of telecommunications, pp. 235–252, 1971.

[11] S. W. Golomb, "Run length codings," IEEE Transactions on Information Theory, vol. 12 no. 7, pp. 399–401, 1966.

[12] R. F. Rice, "Some practical universal noiseless coding techniques," Technical Report 79-22, 1979.
[13] K. Jerbi, M. Raulet, O. Déforges, and M. Abid, "Design of an Embedded Low Complexity Image Coder using CAL language," DASIP 2009 proceeding, September 2009.
[14] N. Siret, M. Wipliez, J.F. Nezan, and A. Rhatay, "Hardware code generation from dataflow programs," DASIP 2010.