A PORTABLE VIDEO TOOL LIBRARY FOR MPEG RECONFIGURABLE VIDEO CODING USING LLVM REPRESENTATION

J. Gorin[1] *- M. Wipliez*[2] *- F. Prêteux*[1] *- M. Raulet*[2]

[1]ARTEMIS, Institut Télécom SudParis, UMR 8145, Evry, France
[2]IETR, INSA Rennes, F-35043, Rennes, France

ABSTRACT

MPEG Reconfigurable Video Coding (RVC) represents the last answer of MPEG to overcome the lack of interoperability between codecs deployed in the market nowadays. The main goal of MPEG RVC is to provide a set of coding tools employed in all MPEG standards, the *Video Tools Library* (VTL), encapsulated into independent entities called *Functional Units* (FUs). FUs are described as dataflow actors in *RVC-CAL actor language* (RVC-CAL) and decoders are described as dataflow programs with the Abstract Decoder Models (ADMs). Therefore, an ADM of an MPEG decoder corresponds in MPEG RVC to a network of FUs taken from the VTL. The typical use of MPEG RVC is to translate an ADM into a hardware or software description language that target one specific platform. In [1], we propose to skip this synthesis process of ADM and to directly integrate a portable version of VTL described in the Low-Level Virtual Machine Intermediate Representation (LLVM IR) inside platforms. This portable VTL is couple with a new RVC Decoder, we called *Just-In-Time Adaptive Decoder Engine* (Jade), that dynamically instantiates ADM to decode any encoded video using its associated network description. In this paper, we introduce the different compiling steps required to obtain an automatically translation of a VTL described in RVC-CAL into a portable VTL described in LLVM. This translation is based on a new RVC-CAL compiler called Open RVC-CAL Compiler (Orcc).

Index Terms— RVC, CAL, LLVM, Orcc, Dataflow programming, Compilation, Intermedia Representation, dynamic decoding.

1. INTRODUCTION

MPEG *Reconfigurable Video Coding* (RVC) has been chosen by the MPEG community to be an alternative paradigm for codec deployment. Its objective is to enable arbitrary combinations of fundamental algorithms, without additional standardization steps. The paradigm used in MPEG RVC is the RVC-CAL Actor Language (RVC-CAL). This language allows developer to produce high-level description of *Abstract Decoder Model* (ADM) by using dataflow programs.

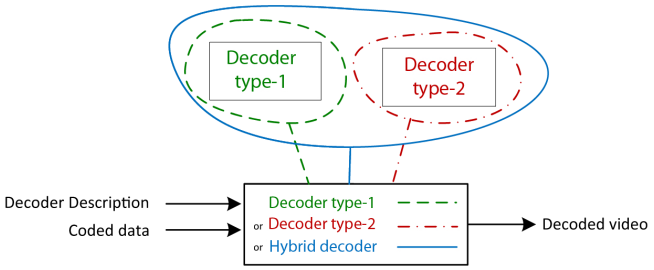

Fig. 1. Representation of the *Just-In-Time Adaptive Decoder Engine*.

An ADM encapsulates the algorithms of an application into independent entities called *Functional Units* (FUs). By using this principle, a dataflow description naturally exposes concurrency between components of an application without adding any implementation details. The MPEG RVC framework provides both a normative standard library of FUs called *Video Tool Library* (VTL) and a set of decoder descriptions expressed as network of FUs. An ADM representation of a decoder is modular and helps its reconfiguration by permitting the topology of its network to be easily modified.

In this context, we developed a new dynamic decoder called *Just-In-Time Adaptive Decoder Engine* (Jade), which follows the concept of MPEG RVC. This decoder is based on the Virtual Machine provided by the *Low-Level Virtual Machine* (LLVM) infrastructure to dynamically load and execute dataflow programs of ADM. This dataflow decoder is created in Jade according to a network description of the ADM and an LLVM representation of the VTL. As represented in Figure 1, Jade is able to take the side information of the decoder description alongside the content itself, for a dynamic generation of decoders 1, 2 or a hybrid version between these two decoders. To be fully RVC compliant, Jade is based on FUs provided by the VTL and translated into LLVM *Intermediate Representation* (IR). This LLVM IR enables a transparent execution of FU on a Virtual Machine.

In this paper, we describe the way to automatically convert a VTL described in RVC-CAL into a Low-Level Virtual Machine equivalent Intermediate Representation (LLVM IR). The LLVM IR is a low-level and platform independent

representation, close to Assembly languages, which provides high-level information for compiler optimizations. By coupling LLVM IR with its associated Virtual Machine, LLVM provides excellent computation performance, comparable to static code execution (i.e. without using a Virtual Machine). After a brief reminder of the concept of MPEG RVC and of its associated language RVC-CAL, this paper presents the different compilation stages starting from an RVC-CAL description of a Functional Unit to obtain an LLVM IR of this FU. This automatic process is helped by a new Open RVC-CAL Compiler (Orcc) that produces the first stage of our compilation process. We will finally conclude this paper with some experiments on size and execution performance of the portable VTL coupled with Jade.

2. MPEG RECONFIGURABLE VIDEO CODING (RVC)

The key approach of MPEG RVC [2] is to produce an *Abstract Decoder Model* of MPEG standards at system-level and suitable for any platform. An ADM is a generic representation of a decoder, built as a block diagram expressed with the *XML Dataflow Format* (XDF). XDF is an XML dialect that describes the connections between blocks (FUs). Each FU is described in RVC-CAL and defines a processing entity of a decoder. Connections represent the data flow between FUs.

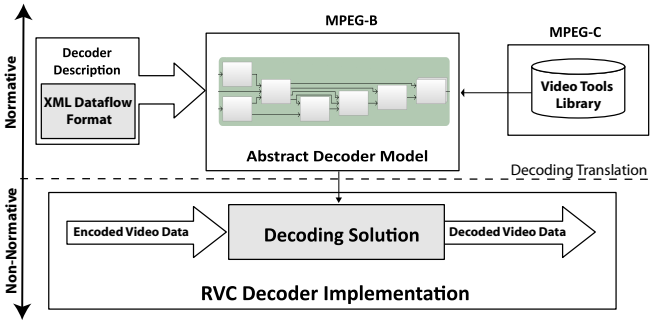

Fig. 2. A typical use of the MPEG RVC Framework.

The MPEG RVC framework [2] is under development as part of the *MPEG-B* standard [3], which describes the framework and the language it uses; and as part of the *MPEG-C* standard [4], which defines the library of video coding tools (*Video Tool Library* or VTL) employed in existing MPEG standards. The Figure 2 shows a typical use of a normative ADM description to produce a non-normative decoding solution that can target either software or hardware platform.

2.1. RVC-CAL dataflow programming

RVC-CAL Actor Language has been chosen by MPEG RVC as the reference programming language for describing FU [5]. An *actor* in RVC-CAL represents an instantiation of an RVC Functional Unit and an RVC-CAL dataflow model represents a composition of *actors*. An actor (shown in Fig. 3 and Fig. 4) is a computational entity with *input ports*, *output ports*, *states*, *actions*, and *parameters*. All actors communicate with others by sending and receiving *tokens* (atomic pieces of data) through their ports.

```
actor Abs () int (size=16) I ==> uint (size=15) O,
                                  uint (size=1)  S :
        pos: action I: [u] ==> O:[u] end
        neg: action I :[u] ==> O:[-u] guard u < 0 end

        unsign: action ==> S:[0] end
        sign:   action ==> S:[1] end

        priority
                neg > pos;
        end

        schedule fsm s0 :
                s0 ( pos    ) --> s1;
                s1 ( unsign ) --> s0;
                s0 ( neg    ) --> s2;
                s2 ( sign   ) --> s0;

        end
end
```

Fig. 3. Description of the **Absolute Value** actor in RVC-CAL.

An actor contains one or several *actions*. Actions define computations that an actor has to execute (or to *fire*). Actions may have a *tag*, *guard*, *local variables*, and *statements*. An action is defined by the amount of tokens consumed on the input. It may change the actor state, and may output tokens according to a function of inputs tokens and state variables. The *guard* conditions specify additional firing conditions, where the action firing depends on the values of input tokens or the current state.

When an actor fires, an action is selected according to the number and value of tokens available, and if the $guard$ associated to the action is $true$. Action selection may be further constrained using a *Finite State Machine* (FSM) and *priority* inequalities to impose a partial order among action tags. The reader can refer to [5] for more details on the RVC-CAL language.

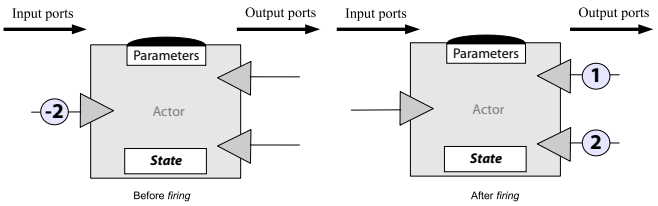

Fig. 4. *Absolute Value* actor before and after *firing*.

An actor called *Absolute Value* is described in RVC-CAL on Figure 3 and is represented on Figure 4. This actor outputs the sign and the absolute value of its input data. It is composed of three ports, one input I of int(size=16) and

two outputs S and O of size respectively uint(size=15) and uint(size=1). The output O produces absolute value of token on input I. The output S produces a token designating the sign of the input data, i.e. 1 if data is negative and 0 otherwise. The actor contains four actions; pos and neg that produce absolute values; $sign$ and $unsign$ for producing signed values. An FSM rules the order in a way that actions can be fired with the sequence *pos* then *unsign* or *neg* then *sign*. The $priority$ favors action *neg* over action *pos* when both are fireable.

3. LOW-LEVEL VIRTUAL MACHINE (LLVM)

RVC-CAL describes dataflow programs at high-level of description without giving any implementation details, exposing available parallelism between the component of an application. On the contrary, the Low-Level Virtual Machine (LLVM) is a low-level representation of applications, close to Assembly languages, that captures the key operations of ordinary processors but avoids machine specific constraints such as physical registers or pipelines. The LLVM *Intermediate Representation* (LLVM IR) has been designed as a low-level representation but with high-level type information for compiler analysis and optimization [6].

The term of LLVM is also designating its associated Virtual Machine capable to manage the LLVM IR of an application for producing highest performance executable code through an aggressive system of continuous optimization.

3.1. LLVM Intermediate Representation

One of the key factors that differentiate LLVM from other systems is the intermediate representation it uses. The reader is invited to read [6] that explains the justification of LLVM IR design choices. The LLVM IR is composed of an infinite set of typed virtual registers that can hold values of primitive types (integral, floating point, or pointer values). These virtual registers are in Three Address Code (3AC) form and Static Single Assignment (SSA) form [6]. We develop the property of these two forms, widely used for compiler optimization, in section 5.1. LLVM programs transfer values between virtual registers and memory solely via *load* and *store* operations using typed pointers.

```
%X = div i32 4, 9 ; Signed int division
%Y = div unsigned i8 12, 4 ; Unsigned char division
%cond = eq i32 %X, 8 ; Produces a bool value
br i1 %cond, label %True, label %False
True:
...
```

Fig. 5. LLVM IR coding example

The LLVM instruction set contains 31 operation codes (*opcode*) that can be overloaded (for example, the *add* instruction can operate on operands of any integer size or vector type). The LLVM IR has also a mechanism for explicit representation of *Control Flow Graph* (CFG). Figure 5 provides an example of the LLVM IR. More information on syntax and semantics of each LLVM instructions are given in the LLVM reference manual [7].

3.2. LLVM inside the Just-In-Time Adaptive Decoder Engine

The Virtual Machine from LLVM represents the core of Jade. Jade is a dynamic RVC decoder able to generate and execute a dataflow program from an ADM, by using its associated decoder description and an LLVM representation of the VTL. The decoder description is a network description of an ADM expressed in XDF. The dynamic decoder creation and execution of jade is made by coupling the Virtual Machine of LLVM with a dedicated library — the *RVC Decoder Engine* (RDE) represented in Figure 6.

The RDE library parses XDF files, parses LLVM IR of required FUs among the VTL, and generates an LLVM representation of the corresponding ADM. The LLVM representation is finally sent to the Virtual Machine that will produce and execute efficient machine code, fitted to the target platform.

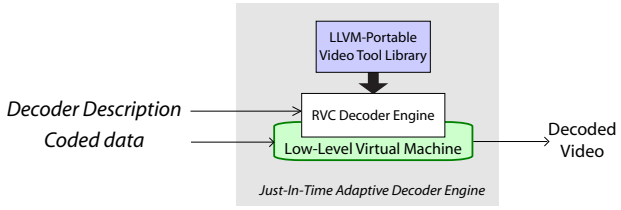

Fig. 6. Infrastructure of Just-In-Time Adaptive Decoder Engine.

The strength of Jade is to take the benefits of the LLVM infrastructure along with the RVC infrastructure. As LLVM is becoming a commercial grade research compiler, the LLVM IR generated from an ADM will continually benefit from improvements of the LLVM compiling infrastructure. The achievement of this solution is also highly related to the efficiency of the LLVM IR of FU in the VTL. The translation of an RVC-CAL FU into LLVM IR must keep the high-level information from the original model, but with a very low-level description of the FU processing and behavior. The next two sections of this article are focused on the LLVM translation process of the VTL.

4. RVC-CAL COMPILER INFRASTRUCTURE

The translation process of an RVC-CAL FU into an llvm-equivalent representation is based on the Open RVC-CAL Compiler ($Orcc$) [8]. Orcc is a compilation framework that

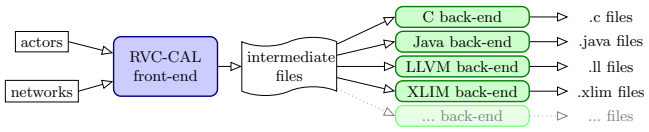

Fig. 7. Open RVC-CAL Compiler Infrastructure.

can translate RVC-CAL programs into a software or hardware representation. This compilation framework is composed of different components, i.e. one front-end and several back-ends. Thus, a compilation process for a given RVC-CAL dataflow program to target a specific language is made in two-steps:

1. A unique front-end parses the FUs of a given network and translates them into a an Orcc-specific Intermediate Representation (Orcc IR),

2. A dedicated back-end loads the network and the actors in Orcc IR form to generate code in the targeted language.

For the need of Jade, we developed a new LLVM back-end that starts from the low-level Orcc IR to produce LLVM IR of an entire VTL.

4.1. Actor Intermediate Representation

The *Orcc IR* is a conservative representation of a dataflow program in terms of structure and semantic while being at a lower level of representation. It is a common denominator of potential target languages such as C, C++ or Java without favoring one in particular.

Each actor representation is serialized in JavaScript Object Notation (JSON) description format. Figure 8 shows the structure of the *Absolute Value* actor (Fig 3) in Orcc IR. The JSON description contains $name$, $pattern$ and a *list of actions* of the actor. The FSM of the actor is translated into an *action scheduler* that lists state-to-state transitions. $Priorities$ become a list of action tags sorted by decreasing priority.

The JSON description of an action (Fig. 9) contains the action tag and the number of tokens produced and consumed. The expressions in the action are reduced to simple arithmetic expressions. Functional tests and list generators become imperative statements. The assignment statement is differentiated into assignments ($assign$) to local variables and $load$/$store$ to memory operations. The high-level RVC-CAL functional expressions containing function calls, conditionals or list generators are translated into an equivalent lower-level IR expression.

The condition to fire an action becomes an $isSchedulable$ function (Fig. 10), which tests value and the number of token on the input of the action, as well as testing $guard$ condition

```
"name": "Abs",
"inputs" : [ "int",  [ 16 ] ], "I" ],
"outputs": [ "uint", [ 15 ] ], "O" ],
"actions": [
[
   (...)
]
"action_scheduler": [
    "s0",  //Initial states
    [ "s0", "s1", "s2"], //States
    [
      [ "s0", [ [ [ "b" ], "s1" ], [ [ "a" ], "s2" ] ] ],
      [ "s1", [ [ [ "n" ], "s0" ] ] ],
      [ "s2", [ [ [ "p" ], "s0" ] ] ]
    ]
  ]
]
```

Fig. 8. Description of the structure of *Absolute Value* actor in Orcc Intermediate Representation.

```
"pos",false,[32,54,5],"void",[],
[
  [["O"],,["List"[1], ["uint",[15]]]],
  [["I"],["List",[1], ["uint",[16]]]],
  [["u"],["uint",[15]]]
],
[
  ["read",[["I"],"I",1]],
  ["load",[["u",1],["I"],[0]]],
  ["store",[["O"],[0],["var",["u",1]]]],
  ["write",[["O"],"O",1]]
]
```

Fig. 9. Description of the action **pos** of *Absolute Value* actor in Orcc Intermediate Representation.

```
"isSchedulable_pos","bool",
[
  [["_tmp",1,1],"bool"],
  [["_tmp",0,1],"bool"],
  [["_tmp",0,2],"bool"],
  [["_tmp",0,3],"bool"]
],
[
  ["hasTokens",[],[["_tmp",1,1],"I",1]],
  ["if",
   ["var",["_tmp",1,1]],
   ["assign",[["_tmp",0,1],[true]]],
   ["assign",[["_tmp",0,3],[false]]]
  ],
  ["join",[["_tmp",0,2],[["_tmp",0,1],["_tmp",0,3]]]],
  ["return",["var",["_tmp",0,2]]]
]
```

Fig. 10. Description of the firing condition of the action **pos** of *Absolute Value* actor in Orcc Intermediate Representation.

if it exists. If $isSchedulable$ returns true, it means that the current action can be fired.

Specific operations dealing with FIFOs become $read$, $write$, $hasTokens$ or $peek$ statements. This is necessary in Orcc IR because the semantics of RVC-CAL specify that the input and output patterns may have read/write several tokens as a list, or may have to reorder tokens.

4.2. Template engine

Each Orcc back-end, which can produce C, $Java$ or $VHDL$ code, has the role to parse Orcc IR from actor into classes, then to unparse these classes to the targeted language using a *template engine*. A template engine is a simple code generator that emits texts using *template documents*. A template engine breaks up template documents into piece of text and attribute expression, producing the grammar and syntax of the functional language. It avoids print statements into Orcc back-end to generate source code and ease the implementation of new language.

A template document is composed of 4 canonical operations:

1. Attribute reference: *$name$*
2. Template references: *$search()$*
3. Apply template to multi-valued attribute: $users:{u — $u.name$ ID is $u.id$.}$
4. Conditional include: *$if(var)$ var is true. $endif$*

These canonical operations are encapsulated into methods corresponding to IR instructions (Fig. 11). Texts or canonical operations include in these methods are printed when a corresponding instruction is found on the Orcc IR.

```
Assign(assign) ::= <<
$assign.target$ = $assign.value$;

>>
```

Fig. 11. Example of a template for assignment instructions in Orcc IR.

A template group can contain several templates, and can extend a parent template. These functionalities allow a template group in a back-end to overwrite templates of a parent template, and thus favor reusability among back-ends. The template engine also supports recursion, automatic indentation, and line wrapping.

5. FROM RVC-CAL COMPILER IR TO LLVM

C, $Java$ or $VHDL$ Orcc Back-ends are designed to produce one targeted-code file per actor and a network file that manage the execution of the overall decoder. These files, representing an ADM, have to be compiled into a single application that can then be installed into the machine we target. For the need of Jade, the llvm back-end does only compile each FU from VTL into a separate LLVM IR file. One VTL generation fits every platform supported by the LLVM infrastructure. The generation and execution of the network is left to the RDE of Jade.

The Orcc IR and LLVM IR have some similarity that helps the translation process for the LLVM back-end. They are both in SSA form with an unlimited number of registers. They have both integer types with arbitrary bit widths. Both IRs have instructions with similar semantics, those include assignment to a local variable, load/store memory operations and ϕ assignments. In the next part, we explain the necessary transformation applied to the Orcc IR to obtain an LLVM IR.

5.1. Three Address Code form

The first main difference between LLVM and the actor IR is that the Orcc IR supports arithmetic expressions in assignments, load/store, and conditional branches, while LLVM only supports three-address code(3AC) [9].

Three-Address Code (3AC) is a form used to improve compiler analysis. Each instruction of the 3AC form is described as a 4-tuple: $operator$, $operand1$, $operand2$, $result$. The general form of 3AC is $x := y\ op\ z$, where x, y and z are variables, constants or temporary variables and op is an arithmetic operator. In Orcc IR, each expression that contains more than one fundamental operation must be decomposed into an equivalent series of instructions that fits 3AC form and SSA constraints. For instance, the instruction $p := x + y \times z$ is converted into $t_1 := y \times z$ and $p := x + t_1$ where t_1 is a variable that is assigned exactly once in the overall function to respect the SSA constraint. Three-address code is still also used even if some instructions use more or fewer than two operands.

The key features of three-address code are that every instruction implements exactly one fundamental operation. Every expression containing more than one fundamental operation in Orcc IR is translated into a series of 3AC instructions before generating the LLVM IR. This transformation is directly applied to the Orcc IR of actors as a transformation pass.

5.2. Structure of template document

The other difference between Orcc IR and LLVM is that an instruction in Orcc IR usually corresponds to many instructions in LLVM IR. For instance, the *hasTokens* node, which checks FIFO status in Orcc IR, is decomposed into a *load* opcode and a *call* opcode in LLVM. To respect the SSA form, the template document of the Orcc back-end is structured into 3 levels of template as shown in Figure 12.

```
//Orcc IR Instructions
HasTokens(hasTokens) ::= <<
  $HasTokensNode(hasTokens.target,
                 hasTokens.numTokens,
                 hasTokens.port)$
>>

// Conversion instructions
HasTokensNode(result, token, port) ::= <<
  $LoadOp(result=result+"0", ty="%fifo**", pointer="@"+port)$
  $CallOp(result=result,
          ty="i1",
          fnptrval="@hasTokens",
          function_args= "%fifo* "+result+"1"+, i32 "+token
          )$
>>

// LLVM Operations
LoadOp(result, ty, pointer) ::= <<
  $result$ = load $ty$* $pointer$
>>

CallOp(result, ty, fnptrval, function_args) ::= <<
  $if(result)$
    $result$ =
  $endif$
  call $ty$ $fnptrval$ (
  $if(function_args)$
    $function_args$
  $endif$
  )
>>
```

Fig. 12. Example of template transformation stage for *hasTokens* instructions.

The first level is devoted to Orcc IR instructions (i.e. *HasTokens*). This template is called by the back-end when Orcc IR has the corresponding instruction. It gets information from the instruction and selects the right transformation to apply. The second level describes the conversion to apply to an instruction (*HasTokensNode*). This template transforms an Orcc IR instruction into a series of LLVM opcodes and ensures that SSA property is respected. The third level manages LLVM opcodes (*LoadOp* and *CallOp*). The overload of each LLVM opcode is managed according to the parameters given to the template. This lower-level of template is the only level that prints LLVM instructions into files.

5.3. Control Flow Graph Transformation

Another main difference between LLVM and Orcc IR is that conditional branch nodes, namely `if` and `while` nodes, have no equivalent in LLVM IR. The Control Flow Graph (CFG) of a function in the LLVM IR is a list of basic blocks, each basic block starting with a label. A basic block contains a list of instructions, and ends with a terminator instruction, such as a branch instruction or function return instruction. We augmented the Orcc IR with a straight Control Flow Graph checking that creates, flattens, and simplifies the CFG.

Figure 13 gives an LLVM-equivalent representation of the scheduler of action *pos* (*isSchedulable_pos*), described in RVC-CAL on Figure 3 and in Orcc IR on Figure 10. This

```
define internal i1 @isSchedulable_pos() {
entry:
  br label %bb1

bb1:
  %_tmp1_10 = load %fifo_s** @I
  %_tmp1_1 = call i32 @hasTokens (%fifo_s* %_tmp1_10, i32 1)
  br i1 %_tmp1_1, label %bb2, label %bb3

bb2:
  br label %bb4

bb3:
  br label %bb4

bb4:
  %_tmp0_2 = phi i1 [ 0 , %bb2 ], [ 1 , %bb3 ]
  ret i1 %_tmp0_2
}
```

Fig. 13. Description of the firing condition of the action **pos** of *Absolute Value* actor in LLVM Representation.

scheduler defines the firing condition of *pos*. It returns 1 if the action *pos* is fireable and 0 otherwise.

The *isSchedulable_pos* LLVM IR function is composed of 5 basic blocs (*entry* and *bb1* to *bb4*) surrounded by *branch* instructions (*br*). It checks the status of the FIFO *I*, by calling *hasTokens* function. The control flow is then branched to *bb1* or *bb2* according to the resulting *%_tmp1_1*. *%_tmp0_2*, corresponding to return value, is set by a *phi* instruction. This instruction takes the value 0 if the previous basic block is bb3, or the value 1 if the previous block is bb2.

5.4. Metadata information

LLVM is also able to manage extensible metadata information inside LLVM IR. Metadata is widely used in programming languages to give "extra" information about some element of the application (for instance variables, functions, structures...).

In our approach, metadata information is used to carry the structural information of an FU. We call structural information of FU the information elements in the Orcc IR that have any influence on computation, namely *name*, *inputs*, *outputs* and *actions* of an FU. This structural information is necessary for Jade to connect actors in dataflow programs and to apply some actor transformation inside an ADM, for instance the merging of actors [10]. Figure 13 corresponds to the metadata description of *Absolute Value* actor. This representation is equivalent to the Orcc IR description given in Figure 8.

The metadata elements are identified in LLVM with a *metadata* type and a preceding exclamation point ('!'). Metadata elements of FUs are explored using *Named metadata*. *Named metadata* are collection of metadata primitives that contain properties of dataflow elements. For instance, the input *I* of *Absolute Value* is described by the named metadata !1 in Figure 14.

Metadata primitives are composed of *metadata strings*

```
!name = !{!0}
!inputs = !{!1}
!outputs = !{!2}
!actions = !{!3, !4, !5, !6}
!action_scheduler = !{!7}

!0 = metadata !{metadata !"Abs"}   ; Name of the actor
!1 = metadata !{ i32 16,           ; Size of the input I
  metadata !"I",                   ; String name of I
  %fifo_s** @I}                    ; LLVM IR variable
      (...)
!3 = metadata  !{metadata !"pos",  ; Tag of action
  metadata !8,                     ; Scheduler of action
  metadata !9}                     ; Body of action
!8 = metadata !{metadata !"bool",  ; Function return type
  metadata !10,                    ; Function pattern
  i1()* @isSchedulable_pos}        ; LLVM function name
!9 = metadata !{metadata !"void",  ; Function return type
       (...)
```

Fig. 14. Description of the structure of *Absolute Value* actor using LLVM *metadata*.

and *metadata nodes*. *Metadata strings* are surrounded by double quotes and give name of dataflow elements (*metadata ¡'I"* for Input I). *Metadata nodes* are presented as comma separated list of elements, surrounded by braces (*!{ i32 16, .., %fifo_s** @I}*). They can have any values, nodes, strings or LLVM variables as operand.

We incorporate LLVM variables inside metadata node to bound metadata information with their corresponding LLVM variables. By encapsulating this information inside the LLVM description, no information is lost between Orcc IR and structural information is directly bounded to the LLVM IR. The representation of the original CAL description is conserved and manageable by the RDE of Jade.

6. GENERATED VIDEO TOOL LIBRARY

We tested our concept of portable VTL on the MPEG RVC VTL at its current state of standardization in MPEG-C [4]. This VTL currently incorporates coding tools from MPEG-4 Part-2 *Simple Profile* (SP) and MPEG-4 *Advanced Video Coding* (AVC) standards. It is composed of *69* RVC-CAL FUs, 22 are coming from the MPEG-4 SP standard and 47 are coming from MPEG-4 AVC standard.

We compared the LLVM back-end with the other back-ends of Orcc namely the C back-end and $Java$ back-end. This comparison puts the focus on the relevance of using LLVM IR as the reference language for Jade into a unique environment. However, C represents the most popular language for static compilation as much as $Java$ is the most popular language for dynamic compilation using a Virtual Machine.

Table 1 compares the size of the generated VTL from these 3 back-ends. All the files include in the resulting VTLs were compiled independently with no optimization enabled. The compiling tools used are *gcc V4.3.2* for C, *javac V1.6.0* for Java and the llvm assembler from LLVM 2.7.

The important size of the C compiled version of VTL is explained by the fact that C files are incorporating FIFO headers, where Java and LLVM doesn't use header and externalize functions. This result shows the lightness of LLVM $bytecode$ size comparable to Java bytecode that allows the *portable VTL* to be easily sent or incorporated into embedded system.

		C	Java	LLVM
	MPEG-4 *SP*	1,74 Mo	300 Ko	285 Ko
+	MPEG-4 *AVC*	2,29 Mo	775 Ko	755 Ko
	VTL	4.03 Mo	1,04 Mo	1,01 Mo

Table 1. Comparison of the compilation size without optimization of a same VTL generated in *C*, *Java* and *LLVM*. The VTL includes FUs from MPEG-4 *Simple Profile* (SP) standard and from MPEG-4 *Advanced Video Coding* (AVC) standard.

The second experimental result puts the focus on the achievement of the generated decoder from each backend. These experiments were led on two configurations of decoders, an RVC ADM of MPEG-4 SP described in [11] and an RVC ADM of MPEG-4 AVC described in [12]. We considered these configurations as the most representative among RVC ADMs as they cover all the VTL.

Table 2 shows the achievement of these 6 generated decoders on an *Intel®E6600 Core™2 Duo* processor at $2.40GHz$ running with *Windows 7™*. The *integrated development environment* (IDE) used to compile generated files from each back-ends were namely *Microsoft Visual Studio 2008 Express* for C, *Eclipse V3.6.0* for JAVA and Jade using *LLVM V2.7* for LLVM.

	C	JAVA	LLVM
MPEG-4 *SP*	26,7 fps	3,5 fps	24,9 fps
MPEG-4 *AVC*	30,9 fps	2,8 fps	31,7 fps

Table 2. Comparison of decoder performance for C, JAVA and LLVM version of the MPEG-4 Part-2 *Simple Profile* configuration on a CIF sequences (352×288), and of the MPEG-4 *Advanced Video Coding* configuration on QCIF sequences (176×144).

These results show that using Jade instead of a statically C-compiled decoder has minor or no impact on the execution of a decoder. On the contrary, Jade is about 7 times faster than the Java version of the same ADM running on the *Java Virtual Machine* (JVM). This speed factor can be explained by the fact that Java, instead of LLVM, has no pointer. FIFO accesses in JAVA involve a memory copy of data, which puts pressure on the garbage collector and reduces the decoder performance.

Jade has also been tested with this same test environment, i.e. same decoder configurations and an only version

the portable VTL in LLVM IR, on different *Operating Systems* (OSes) (*Linux* and *MAC OS X*) and different platforms (*ARM*® V7 and *PS3*®). The portable VTL used were compiled once in a single environment, and then copied into each tested platform. The resulting performance of Jade on these two configurations is highly dependent on computation efficiency of each platform, but proves the portability of our LLVM-generated VTL.

7. CONCLUSION AND PERSPECTIVES

This paper describes the compilation stage required to translate RVC-CAL description of FUs from the VTL of MPEG RVC to generate a portable VTL, efficiently expandable into a wide variety of platforms. The generation of this portable VTL is the most crucial elements for the achievement of our RVC decoder called Jade. We proved the relevance of using the LLVM IR for this portable Video Tool Library by comparing achievement of equivalent decoder descriptions with the two most popular languages used nowadays in decoding systems. By using dynamic property of LLVM, Jade represents an excellent starting point for new research on adaptive decoding.

However, the Jade framework is still a base for many works. Its *Reconfigurable RVC Engine* (RDE) does not yet take into account many information included in the portable VTL. For instance changing a decoding solution in another decoding solution needs Jade to be fully reconfigured. We plan to reduce this reconfiguration time by finding parts of a dataflow decoder that really need to be changed (i.e. partial reconfiguration). Using CAL dataflow programming involves scalable parallelism adapted from uni-core systems to massively multi-core systems, but a smart dispatching of decoding algorithms onto different processing elements has to be implemented into the RDE. Finally, we take MPEG RVC as the base application for Jade, but the concept of ADM representation can also be extended to others medias that involve signal processing, such as audio, cryptographic or 3D applications.

8. REFERENCES

[1] J. Gorin, M. Wipliez, J. Piat, F. Preteux, and M. Raulet, "An LLVM-based decoder for MPEG Reconfigurable Video Coding," in *Proc. of IEEE ICC 2010*. October 2010, IEEE.

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

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

[4] ISO/IEC FDIS 23002-4, *MPEG systems technologies – Part 4: Video Tool Library*, 2010.

[5] I. Amer, C. Lucarz, M. Mattavelli, G. Roquier, M. Raulet, O. Déforges, and J.F. Nezan, "Reconfigurable Video Coding: An overview of its main objectives.," in *IEEE Signal Processing Magazine, Special Issue on Signal Processing on Platforms with Multiple Cores*, 2009.

[6] C. Lattner, "LLVM: An Infrastructure for Multi-Stage Optimization," M.S. thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002, *See* `http://llvm.cs.uiuc.edu`.

[7] C. Lattner and V. Adve, "LLVM Language Reference Manual," Tech. Rep. 4th edition, ECMA International, June 2006.

[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, NY, USA, 2010, pp. 223–234, ACM.

[9] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[10] M. Wipliez and M. Raulet, "Classification and Transformation of Dynamic Dataflow Programs," in *The 2010 Conference on Design and Architectures for Signal and Image Processing (DASIP 2010)*. 2010, IEEE Computer Society.

[11] 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.

[12] J. Gorin, M. Raulet, Y.L. Cheng, H.Y. Lin, N. Siret, K. Sugimoto, and G.G. Lee, "An RVC Dataflow Description of the AVC Constrained Baseline Profile Decoder," in *Proceedings of ICIP'09*, Nov. 2009.