LLVM-based and scalable MPEG-RVC decoder

Jérôme Gorin, Matthieu Wipliez, Françoise Préteux, Mickaël Raulet

Springer manuscript No.
(will be inserted by the editor)

Jérôme Gorin · Matthieu Wipliez · Françoise Prêteux · Mickaël Raulet

LLVM-based and scalable MPEG-RVC decoder

Received: date / Accepted: date

Abstract MPEG *Reconfigurable Video Coding* (RVC) is a new platform-independent specification methodology chosen by the MPEG community for describing coding standards. This methodology aims at producing Abstract Decoder Models (ADMs) of MPEG decoders as programs described in a dataflow language namely "*RVC-CAL Actor Language*" (RVC-CAL). RVC-CAL naturally expresses potential parallelism between tasks of an application, which makes an ADM description suitable for implementation to a wide variety of platforms, from uniprocessor systems to FPGAs. MPEG RVC eases the development process of decoders by building decoders at a library-component level instead of using monolithic algorithms, and by providing a library of coding tools standardized in MPEG. This paper presents new mechanisms based on the Low Level Virtual Machine (LLVM) that allow the conception of a decoder able to dynamically instantiate several RVC decoder descriptions. This decoder, unlike static decoders generated by RVC tools, keeps *de facto* the features of an RVC description namely portability, scalability and reconfigurability.

Keywords Reconfigurable Video Coding · RVC-CAL Actor Language · Low Level Virtual Machine · Network scheduling · Dataflow programming · Code synthesis · Low Level Virtual Machine · Multi-core systems

Jérôme Gorin
ARTEMIS, Institut Télécom SudParis, UMR 8145, Evry, France
E-mail: Jerome.Gorin@it-sudparis.eu

Matthieu Wipliez
IETR, INSA Rennes, F-35043, Rennes, France
E-mail: Matthieu.Wipliez@insa-rennes.fr

Françoise Prêteux
ARTEMIS, Institut Télécom SudParis, UMR 8145, Evry, France
E-mail: Francoise.Preteux@it-sudparis.eu

Mickaël Raulet
IETR, INSA Rennes, F-35043, Rennes, France
E-mail: mickael.raulet@insa-rennes.fr

1 Introduction

In the last two decades, the evolution of video coding standards has produced outstanding results. The next challenge of video coding is to speed up adoption of future coding techniques and to make smoother implementation of multiple standards support for a single platform. So as to overcome the limitation of the current monolithic MPEG decoder reference description, the MPEG consortium decides to create MPEG *Reconfigurable Video Coding* (RVC) [17]. This framework aims "at providing a framework allowing a dynamic development, implementation and adoption of standardized video coding solutions with features of higher flexibility and reusability"

Reconfigurable Video Coding (RVC) has been chosen by the MPEG community to be an alternative paradigm for codec deployment. The MPEG RVC paradigm is based on RVC-CAL Actor Language (RVC-CAL) to describe decoders at high-level library component using dataflow descriptions. The main objective of RVC is to enable arbitrary combinations of fundamental algorithms, without additional standardization steps. By adding the side-information of the combination description alongside the content itself, MPEG RVC defines the new concept of RVC decoder. An RVC decoder may create, configure and re-configure video compression algorithms adaptively to its content.

Yet, no mechanism has been found to automatically and dynamically use dataflow description to form an RVC decoder. This paper proposes to transform the RVC-CAL description of coding tools into the generic low-level description called Low-Level Virtual Machine (LLVM) Intermediate Representation (IR), and to use the LLVM infrastructure for dynamically and efficiently instantiating these coding tools to create decoders. By combining the LLVM and the RVC concepts, we created a *portable* MPEG decoder engine that can configure and reconfigure an MPEG RVC decoder description. After a brief reminder of the MPEG RVC and CAL concepts in

section 2, we present the motivation for using LLVM as the reference IR for CAL actor in section 3. The section 4 and 5 are devoted to the implementation of our proposed dynamic RVC decoder. We finally show in section 6 that the decoder has been successfully tested onto multiple platforms.

2 MPEG Reconfigurable Video Coding

The MPEG Reconfigurable Video Coding (RVC) [17] framework is a new ISO standard still under consideration in MPEG. It aims at providing video codec specifications at the level of library components instead of monolithic algorithms. The key approach of MPEG RVC is to produce an *Abstract Decoder Models* of MPEG standard suitable for any platform. An ADM is a generic representation of a decoder, built as a dataflow diagram expressed with the *XML Dataflow Format* (XDF). XDF is an XML dialect that describes the connections between Functional Units (FUs). Each FU is described in RVC-CAL Actor Language (RVC-CAL) and defines a processing entity of a decoder. Connections in an XDF diagram represent the data flow between FUs.

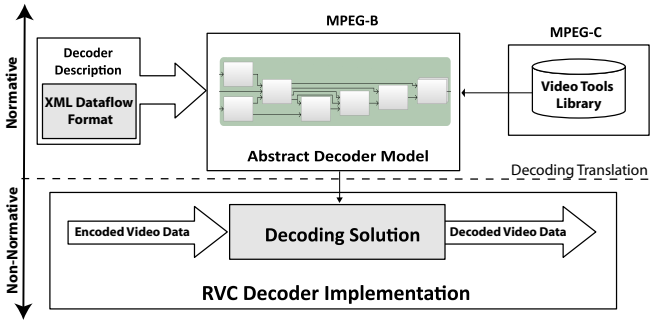

Fig. 1 A typical use of the MPEG RVC Framework.

The Figure 1 shows a typical use of a normative ADM description to produce a non-normative decoding solution that can target either software or hardware platform. MPEG RVC provides both a normative standard library of FUs called Video Tools Library (VTL) and a set of decoder descriptions/configurations expressed as networks of FUs. Such a representation is modular and helps the reconfiguration of a decoder by modifying the topology of the network. Adding new coding technologies in an existing standard is a particularly sensitive part of a standardization process. RVC mainly focuses on reusability of standardized coding tools by allowing different decoder descriptions to instantiate common FUs across standards.

2.1 RVC-CAL Dataflow Programming

RVC-CAL [4] has been chosen by MPEG RVC as the reference programming language for describing FUs. RVC-CAL is designed to provide a concise and high-level description of actors. An *actor* in RVC-CAL represents an instantiation of an RVC Functional Unit and an RVC-CAL dataflow model represents a composition of *actors*. RVC-CAL, compared with the CAL Actor Language (CAL) [8], restricts the data types, and operators that cannot be easily implemented onto the platforms.

An actor shown in Figure 2 and Figure 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. 2 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 input 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*

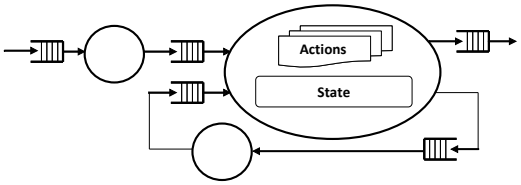

Fig. 3 A CAL network with details on an actor.

(FSM) and *priority* inequalities to impose a partial order among action tags. The reader can refer to [4] for more details on the RVC-CAL language.

A composition of RVC-CAL actors forms a CAL network, represented in Figure 3. In an RVC-CAL network, actors communicate via unbounded channels by reading and writing tokens from and to FIFOs. At a network level, each actor works concurrently, executing their own sequential operations. Dataflow programming was once invented to address the problem of parallel computing. The strength of using dataflow model is that execution of concurrent processes is only driven by token availability. Actors are able to fire simultaneously regardless of the environment, allowing the application to be easily distributed over different processing elements. This feature is particularly useful in the context of multi-core platforms.

Another important point of RVC-CAL is that its dataflow network does not specify any execution models. Many variants of Dataflow Model of Computation (MOC) based on restriction of dataflow models (such as Synchronous Dataflow, Boolean Dataflow...) have been introduced in the literature [15,21,6,3,5]. RVC-CAL is expressive enough to specify a wide range of programs that follow a variety of dataflow models, trading between expressiveness (the set of programs that can be modeled) and analyzability. RVC-CAL dataflow program fits all of those models depending on the environment and the targeted application.

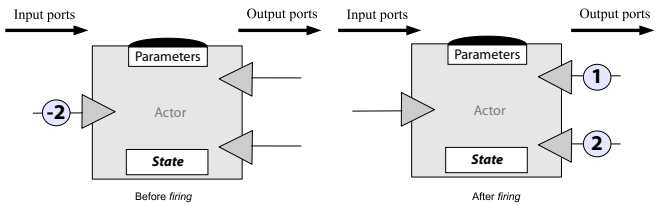

Fig. 4 *Absolute Value* actor before and after *firing*.

Figure 2 shows a simple multiplexer in accordance with RVC-CAL semantics. This multiplexer is composed of three inputs, one output and two actions. Guards prevent to fire the first action when a token on input *c* is not *true* and to fire the second action if token on input c is not *false*. The actor in Figure 4 is an execution example of this multiplexer.

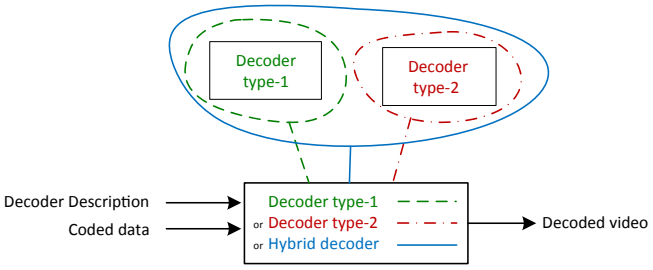

Fig. 5 Representation of the dynamic MPEG RVC decoder.

2.2 MPEG RVC Framework

CAL is supported by a portable interpreter infrastructure that can simulate a hierarchical network of actors. This interpreter was first used in the Moses[1] project. Moses features a graphical network editor, and allows the user to monitor actor's execution (actor state and token values).

The project being no longer maintained, it has been superseded by the Open Dataflow environment (OpenDF[2] for short). OpenDF is also a compilation framework with a backend for generation of HDL(VHDL/Verilog) [11], and a backend targeting ARM11 and embedded C is under development [18] as part of the EU project ACTORS[3]. It is also possible to simulate CAL models in the Ptolemy II[4] environment.

Contrary to aforementioned tools dedicated to CAL, the Open RVC-CAL Compiler[5] (Orcc) is a compiler specific to RVC and its subset of CAL standardized as RVC-CAL. This framework is based on a platform-agnostic, language-agnostic Intermediate Representation (IR) that is both higher-level, more conservative, and more concise than the existing IR used by OpenDF. This IR provides a sort of "common ground" between software languages (C/C++, Java) and Hardware Description Languages (HDLs) such as Verilog and VHDL. The Orcc IR is more detailed in section 4.1. Orcc has backends that can transform the IR to C, C++, Java; backends for VHDL and XLIM are being developed. This tool is available as an Eclipse feature and comes with an IDE composed of a simulator, a debugger, and an editor with automatic error reporting.

3 Low level Virtual Machine Infrastructure

MPEG RVC defines an abstract description of decoders that enhance parallelism scalability, modularity and generic uses. It also provides synthesis tools that generate static decoder programs for specific platforms. The

[1] http://www.tik.ee.ethz.ch/ moses
[2] http://opendf.sourceforge.net
[3] http://www.actors-project.eu
[4] http://ptolemy.eecs.berkely.edu
[5] http://orcc.sourceforge.net

generated decoders are made with information from an ADM description to fully exploit processing resources but reject unnecessary information for the selected platform.

The main contribution of this article is to propose a dynamic synthesis and execution of an ADM description integrated inside the targeted platform. It permits to bring all information from an ADM description to the decoder and to use a Virtual Machine dedicated to RVC ADM for dynamically executing the corresponding decoder. As represented on Figure 5, this decoder may take the side information of the ADM alongside the content itself, for a dynamic generation of decoders 1, 2 or a hybrid version between these two decoders.

Although an actor defined in RVC-CAL looks very compact and clear, writing a compiler for the RVC-CAL language is a nontrivial task, and it takes a lot of workforce to build one from scratch [24]. The *Universal Video Decoder* (UVD), a Virtual Machine dedicated to video decoding with a low-level and specific language, has already proven the benefits of using dynamic decoding in [20]. This concept has shown interesting decoding performance but at the cost of developing a dedicated Virtual Machine, only tested and specified in one specific environment. As for target code portability, we believe that using an existing infrastructure (such as the Java Virtual Machine (JVM) or the Common Language Runtime (CLR)) is mandatory, rather than developing one from scratch.

Performance and portability is a crucial point for the choice of a Virtual Machine. High Level Language Virtual Machine that target dynamic language, such as JVM for Java or CLR for C#, remains more than twice slower than the equivalent C application [22, 23, 13, 2]. The Low-Level Virtual Machine fits all of our expectation by providing excellent application performance, a compiler infrastructure tested in a wide variety of platforms and a strong research infrastructure support.

3.1 Intermediate Representation

One of the key factors that differentiate LLVM from other systems is the intermediate representation it uses. 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. It has been designed to be a low-level representation but with high-level type information for compiler analysis and optimization. The reader is invited to read [13] 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 [13]. We develop the property of these two forms, widely used for compiler optimization, in section 4.2. LLVM programs transfer values between virtual registers and memory solely via *load* and *store* operations using typed pointers.

```
%X = add i32 4, 9 ; affect 4 added to 9 in %X
%Y = call i32 @hasToken() ; Call hasToken function
%cond = eq i32 %Y, 1 ; Produces a bool value
br i1 %cond, label %True, label %False ; Cond. branch
True:
...
```

Fig. 6 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 6 provides an example of the LLVM IR. More information on syntaxes and semantics of each LLVM instructions are given in the LLVM reference manual [7].

3.2 Just-In-Time Compiler

LLVM also designates a compilation framework that exploits the LLVM IR to provide a combination of features not available in any previous compilation approach[14]. These capabilities are:

1. *Persistent program information*: The compilation model preserves the LLVM representation throughout an application's lifetime, allowing sophisticated optimizations to be performed at all stages of execution.
2. *Transparent runtime model*: The system does not specify any particular object model, exception semantics, or runtime environment, thus allowing any language to be compiled using it.
3. *Uniform, whole-program compilation*: Language-independence makes it possible to optimize and compile all code comprising an application in a uniform manner.

The compilation framework corresponds to a collection of libraries and tools that makes easier to build *offline* compilers, optimizers or *Just-In-Time* (JIT) code generators. LLVM is currently supported on X86, X86-64, PowerPC 32/64, ARM, Thumb, IA-64, Alpha, SPARC, MIPS and CellSPU architectures. As LLVM is becoming a commercial grade research compiler, the code generated will continually benefit from improvements of its compiling infrastructure.

3.3 LLVM-based dynamic decoder

The advantages of LLVM are the properties looked for MPEG RVC decoders: generic, efficient and dynamic. As represented in Figure 7, the LLVM JIT Compiler represents the core of the proposed dynamic MPEG RVC decoder. The LLVM is surrounded by two RVC-specific components that bring LLVM compliant with RVC ADM. The first component corresponds to an equivalent-representation of a VTL provided by MPEG RVC, but is described into an LLVM representation. This VTL, we call it portable VTL, must keep the same information from an RVC-CAL actor with a lower level of representation of its computation to be manageable by LLVM. The section 4 is dedicated to the production of this portable VTL.

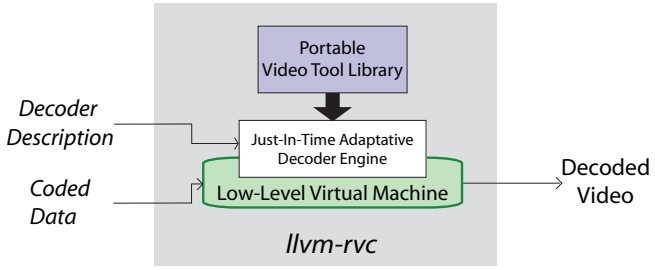

Fig. 7 Infrastructure of our dynamic RVC decoder.

The second component works as a layer of the LLVM JIT Compiler. The Just-In-Time Adaptive Decoder Engine (JADE) manages a description/configuration of an ADM and the portable VTL to produce decoders in LLVM IR. This LLVM IR of ADM is finally sent to the JIT Compiler to produce efficient machine code, fitted to the target platform. JADE is also able to produce network scheduling instructions and to manage the execution of the final decoder. The section 5 is dedicated to the description of this engine.

4 LLVM Code Generation of CAL Actor

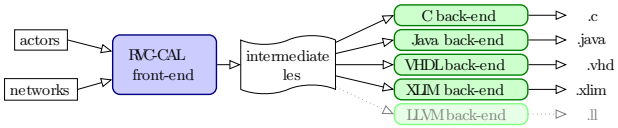

Fig. 8 Infrastructure of *llvm-rvc* decoder.

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. We choose to base the translation process for the portable VTL –i.e. from an RVC-CAL FU of the VTL into an LLVM-equivalent representation– on the Orcc Intermediate Representation (Orcc IR) as this representation is closer to the LLVM IR. The compilation framework of Orcc 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.

To produce the portable VTL needed by the 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 Orcc 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 9 shows the structure of the *Absolute Value* actor (Fig 2) 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 a list of state-to-state transitions. *Priorities* become a list of action tags sorted by decreasing priority.

```
"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. 9 Description of the structure of *Absolute Value* actor in Orcc Intermediate Representation.

The JSON description of an action, provides in Figure 10, 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.

```
"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. 10 Description of the action **pos** of *Absolute Value* actor in Orcc Intermediate Representation.

The conditions to fire an action become an *isSchedulable* function (Fig. 11), which tests value and the number of token on the input of the action, as well as testing *guard* condition if it exists. If *isSchedulable* returns true, it means that the current action can be fired.

```
"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. 11 Description of the firing condition of the action **pos** of *Absolute Value* actor in Orcc Intermediate Representation.

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 LLVM Transformation

The llvm back-end produces a portable VTL by translating each FU of the VTL in Orcc IR into separate files in LLVM IR. One VTL generation fits every platform supported by the LLVM infrastructure. The Orcc IR and LLVM IR have some similarities that help 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.

The first main difference between LLVM and the Orcc IR is that the Orcc IR supports arithmetic expressions in assignments, load/store, and conditional branches, while LLVM only has support for three-address code(3AC) [14]. 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*, *operand*1, *operand*2, *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 is decomposed into an equivalent series of instructions fit to 3AC form and SSA constraint.

```
define void @pod() {
entry:
  ;Read node
  %I = call i8* @getReadPtr (%fifo* @I, i32 1)

  ;Load node
  %u_0 = getelementptr i8* %I, i1 0
  %u_1 = load i8* %u

  ;Write node
  %O = call i8* @getWritePtr (%fifo* @O, i32 1)

  ;Store node
  store i8 u_1, i8* %O
  ret void
}
```

Fig. 12 Description of the action **pos** of *Absolute Value* actor in LLVM Representation.

The other 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.

The Figure 12 represents the body of the action *pos*, described in CAL on Figure 2 and in Orcc IR on Figure 10. Its associated *firing condition* is represented in LLVM on Figure 13 and in Orcc IR on Figure 11.

LLVM Metadata information is used to carry the structural information of an FU. Structural information of an FU is the elements in the Orcc IR that have any influence on computation, namely *name*, *inputs*, *outputs*, *parameters*, *actions* and *FSM* of an FU. This structural

```
define i1 @isSchedulable_pos() {
entry:
  %_tmp1_0 = call i1 @hasTokens (%fifo* @I, i32 1)
  br i1 %_tmp1_1, label %bb2, label %bb3

bb2:
  br label %bb4 ; Fifo has token

bb3:
  br label %bb4 ; Fifo has no token

bb4:
  %_tmp0_1 = phi i1 [1 , %bb2], [0, %bb3]
  ret i1 %_tmp0_1
}
```

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

information is necessary for the JADE to connect actors in dataflow programs, to generate schedulers or to apply actor transformation inside an ADM, for instance the merging of actors [25]. Figure 14 corresponds to the metadata description of *Absolute Value* actor. This representation is equivalent to the Orcc IR description given in Figure 9.

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

; Name of the actor
!0 = metadata !{metadata !"Abs"}

;Description of I
!1 = metadata !{ i32 16,  ; Size
  metadata !"I",  ; Name
  %fifo_s** @I}  ; LLVM variable
      (...)

;Description of pos
!3 = metadata  !{metadata !"pos",  ; Tag
  metadata !8,  ; isSchedulable function
  metadata !9}  ; Body function

;Description of isSchedulable_pos
!8 = metadata !{metadata !"bool",  ; Return type
  metadata !10,  ; Pattern
  i1()* @isSchedulable_pos}  ; LLVM function

;Description of body
!9 = metadata !{metadata !"void",
       (...)
```

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

LLVM variables and functions are incorporated inside metadata to bound metadata information with their corresponding elements in the LLVM IR. Using metadata allows to have no information lost between Orcc IR and an LLVM IR of an actor.

5 Just-In-Time Adaptive Decoder Engine

The second step of the approach is to produce a decoder that dynamically instantiates MPEG decoders according to a network description/configuration and the portable VTL. To achieve this goal, the Just-In-Time Adaptive Decoder Engine manages the LLVM infrastructure for dynamically translating LLVM IR into optimized machine code. The JADE also integrates scheduling rules to execute the generated decoder.

5.1 Decoder Code Generation

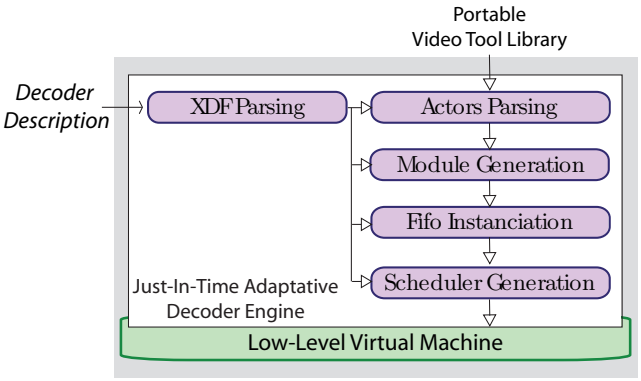

Fig. 15 Infrastructure of the Just-In-Time Adaptive Decoder Engine.

The role of JADE is to manage the instantiation of networks. Instantiation of a network designates the initialization of actors, FIFOs and the network scheduling from an RVC description. JADE creates LLVM representation of decoders in a five steps process described on Figure 15 :

1. *XDF parsing*: XDF description is parsed into a Directed Acyclic Graph (DAG) that describes the structure of the network. This DAG helps analysis based on dataflow graph as explain in [12]. XDF parsing also provides information about the list of FUs in the VTL, the number of instances of an FU and connections in the network to the *actor parsing*, *FIFO instantiation* and *scheduler generator*.
2. *Actor parsing*: Required FUs in the VTL are parsed using LLVM metadata to keep information about actor behavior and structure. They are stored in memory in a form of simple structure that will be sent to the *module generation*.
3. *Module generation*: The parsed FUs are duplicated in an only LLVM module that represents the LLVM representation of the decoder. This is necessary because some FUs can be instantiated many times. It also helps the LLVM JIT Compiler to process optimization on the overall decoder.

4. *FIFO instantiation*: FIFOs are instantiated and connected to every actor's ports. The FIFOs used are unidirectional circular buffers that allow actors to be executed on different threads without the use of semaphores for data synchronization.
5. *Scheduler generation*: Scheduling instructions are inserted in the decoder module to define how processes are given access to system resources. The scheduling part of the decoder has been designed to be modular and interchangeable to give the choice to developers to select scheduling algorithms that best fit the targeted platform.

The generated decoder, represented as an only LLVM module, is finally sent to the LLVM JIT Compiler to produce efficient machine bytecode of the decoder in a three steps process:

1. *Optimization*: The generated single module passes through aggressive optimizations, selected according to their efficiency for the targeted platform.
2. *Compilation*: LLVM module is JIT translated into *machine bytecode*. If the targeted platform is not supported by the JIT Compiler, LLVM also integrates an interpreter that executes LLVM IR line by line.
3. *Execution*: Main scheduler function is executed in a dedicated thread that can be run, stop and partially reconfigured at any times by the JADE.

5.2 Decoder execution

Network scheduling process corresponds to the most sensitive part of JADE for decoder's achievement. For the first implementation of the JADE, we choose to implement a simple round-robin scheduler that avoids complex graph analysis and offers abilities for multi-core execution [1]. This round robin scheduler follows the *Dataflow Process Networks* (DPN) model introduced by [16]. DPN models are dynamically scheduled; hence this scheduler consists in an endless call of each actor in the network as represented in Figure 16.

```
define void @scheduler() {
entry:
  call void @actor_1
  call void @actor_2
  .
  .
  call void @actor_n
  br label %entry ;loop to entry
}
```

Fig. 16 LLVM representation of a round robin scheduler for a network of n actors.

An actor is called by executing its associated *action scheduler*. An action scheduler is a function added to each actor at *scheduler generation* that rules the firing of actions. Figure 17 represents the action scheduler of the actor *pos*. The Finite State Machine (FSM) of an RVC-CAL Actor is translated into an LLVM *switch* instruction. The switch instruction calls a *state scheduler* corresponding to the current actor state. Figure 18 represents the state scheduler of state *s1* in actor *abs*. Actions are listed according to their priorities in the FSM. Firing an action corresponds to test if an action is fireable, namely by calling the associated *isSchedulable_* function. If an action is defined as fireable, the function corresponding to the body of the action can be called.

```
define void @abs() {
entry:
  switch i32 @_FSM_state, label %default [
    i32 0, label %s0
    i32 1, label %s1
    i32 2, label %s2
  ]

s0:
  %s0 = call i1 @s0_scheduler ()
  br i1 %s0, label %entry, label %return

s1:
  %s1 = call i1 @s1_scheduler ()
  br i1 %s1, label %entry, label %return

s2:
  %s2 = call i1 @s1_scheduler ()
  br i1 %s2, label %entry, label %return

return:
  ret void
}
```

Fig. 17 Representation of the action scheduler of the actor *abs* in LLVM IR.

```
define i1 @s1_scheduler() {
entry:
  %0 = call i1 @isSchedulable_neg()
  br i1 %0, label %fire_neg, label %skip_neg

fire_neg:
  call void @neg()
  store i32 s2, i32* @_FSM_state
  ret i1 1

skip_cmd_newVop:
  %1 = call i1 @isSchedulable_pos()
  br i1 %1, label %fire_pos, label %return

fire_cmd_textureOnly:
  call void @pos()
  store i32 s1, i32* @_FSM_state
  ret i1 1

return:
  ret i1 0
}
```

Fig. 18 Representation of the scheduler of the state *s0* for the actor *abs* in LLVM IR.

The strength of the round-robin scheduler consists in the preservation of the scalability of the original model. Indeed, all actors are still considered as independent entities, without any knowledge about the execution se-

quence of actions. The execution of the network –i.e. calls of action schedulers – can be made by one or more round robin scheduler. A round robin scheduler can contain from one to all the actors of a network. In a case of multiple cores, a group of FUs can be mapped onto several round robin schedulers and each round robin scheduler can be affected onto a separate core of the platform. This scheduler enhances parallelism and pipelining in each process of the whole application.

The strength of our model constitutes also its principal weakness if we compare the achievement of our decoder with an equivalent sequential code for a uniprocessor system. As the scheduler does not contain any information about the order to execute actions for each actor, every action has to be tested before determining if it can be executed. This scheduler involves an important overhead that can be reduced by finding static execution rules in the original dataflow network. We plan in future works to reduce this overhead by coupling our scheduler with analyzing tools for dataflow networks that would automatically detect sequential execution order of actions as presented in [10,25]. LLVM metadatas that we embedd inside LLVM IR of actor enable at runtime such analysis.

6 Results

This section presents the experimental results led on the proposed concept. The goals of these experiments are to show the portability of the VTL, the relevance of using LLVM as an intermediate representation and the multicore ability of the scheduler.

With this aim in mind, we developed the JADE, described in Figure 5, in C++[6] coupled with LLVM 2.6[7]. We also used the Orcc LLVM backend to generate a portable VTL from RVC-CAL standardized FUs.

The FUs currently available in the VTL of MPEG RVC are the coding tools from MPEG-4 Part-2 *Simple Profile* (SP) and MPEG-4 *Advanced Video Coding* (AVC) standards. The JADE has been compiled and the portable VTL copied on several OS with an Intel E6600 Core2 Duo processor at 2.40 GHz running with *Windows 7*, *Mac OS X 10.5* and *Linux Ubuntu 9.10*.

6.1 Portable VTL generation

The portable VTL has been tested on the MPEG RVC VTL at its current state of standardization. This VTL is currently composed of *69* RVC-CAL FUs, 22 are coming from the MPEG-4 SP standard and 47 are coming from MPEG-4 AVC standard. The portable VTL generated is compared on table 1 with the resulting files of other backends of Orcc, namely the *C* back-end and *Java* back-end, running on the same VTL. The compiling tools used are *gcc V4.3.2* for C, *javac V1.6.0* for Java and the llvm assembler from LLVM 2.6.

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

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 given results shows the lightness of LLVM *bytecode* size comparable to Java bytecode, which allows the *portable VTL* to be easily sent or incorporated into embedded system. All the files include in the resulting VTLs were compiled independently with no optimization enabled. 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 does not use header and externalize functions.

6.2 Real-Time decoding of CIF video

The second experiment compares the achievement of decoders generated by JADE with equivalent static C and Java decoders with the same round-robin scheduling execution. The configurations of decoders used are described in [4] for the MPEG-4 Part-2 *Simple Profile* (SP) decoder and from [9] for the MPEG-4 *Advanced Video Coding* (AVC) *Constrained Baseline Profile* (CBP). We considered those two configurations of decoder as the most representative of MPEG RVC as they cover all the VTL. Some of these FUs are instantiated several times in decoders, the RVC description of the MPEG-4 SP decoder contains 64 instances and the RVC description of the MPEG-4 AVC contains 92 instances. These decoders were respectively tested on conformance testing sequences for SP decoders[8] and AVC decoders[9].

	C	Java	LLVM
Windows (VS C++)	26,7 fps	7,0 fps	24,9 fps
Linux/MacOsX (GCC)	26,6 fps	9,7 fps	26,4 fps

Table 2 Decoder performance of an MPEG-4 Part-2 *Simple Profile* configuration for CIF sequences (352 × 288).

[6] C++ implementation of Jade is available at: http://sourceforge.net/projects/orcc

[7] LLVM 2.6 is available at: http://llvm.org/

[8] Video sequences available at: http://standards.iso.org/

[9] Video sequences available at: http://wftp3.itu.int/av-arch/jvt-site/

The *integrated development environment* (IDE) used to compile C and Java is namely *Microsoft Visual Studio 2008 Express* for C and *Eclipse V3.6.0* for JAVA. The comparison results, on Table 2 and Table 3, shows that the impact of the LLVM Virtual Machine is unseen on Windows and Linux. The static Java versions of these same decoders running on the *Java Virtual Machine* (JVM) are running three times slower than LLVM decoders. This speed factor can be explained by the fact that does not allocate local arrays on the stack. Each fifo access involve new memory allocation, which must be freed by the garbage collector of the Virtual Machine.

	C	Java	LLVM
Windows (VS C++)	30,9 fps	5,6 fps	31,7 fps
Linux/MacOsX (GCC)	34,9 fps	5,5 fps	34,5 fps

Table 3 Decoder performance of an MPEG-4 *Advanced Video Coding* configuration for QCIF sequences (176 × 144).

Preliminary results show that the configuration and reconfiguration times of the MPEG-4 SP decoder are about 800 milliseconds and about 1 second for the MPEG-4 AVC description. This reconfiguration time can be greatly improved by encapsulating dataflow network representation into bitstream and by supporting partial reconfiguration of decoders.

6.3 Scalable execution

This section tests the multi-core abilities of JADE generated decoders on the Core2Duo processor. The use case are taken from [1] and applied to the Jade. Two POSIX threads were used on *Linux* to implement two round-robin schedulers, with one round-robin scheduler per core. Each scheduler was precompiled with a list of actors manually distributed. The decoders used is the same configuration of MPEG-4 Part-2 *Simple Profile* (SP) presented on Table 2.

Core 1	Core 2	gain
Parser	Texture + Motion	1%
Parser + Motion	Texture	33%
Parser + Texture	Motion	50%

Table 4 Gain by an execution on two threads of the MPEG-4 *Simple Profile* decoder compared to a unicore execution.

The performance given on Table 4 shows benefits of the multi-core (up to 1.5), depending on the configuration used. We considered these results as a proof that a network analysis is mandatory to obtain a smart and automatic dispatchment of the actors into several schedulers.

7 Conclusion and perspective

This paper presents a new concept of dynamic decoder able to fully support ADM from MPEG RVC. It enables an automatic translation of dataflow programs written in RVC-CAL into an LLVM representation to take the benefits of the LLVM infrastructure. As a result, generated decoder can be dynamically compiled and executed with reasonable impact on the performance, relatively with a static compilation of the same decoder. Moreover, the generated decoder contains scalable parallelism based on a simple mechanism that can be introduced into a wide range of platform.

The next milestone of this concept of dynamic RVC decoder is to get an automatic distribution of the actors from a dataflow program into the processing resources of the targeted platform. *The Dif package* [19] shows some mechanisms to insert more information about decoder execution, specifically static execution of part of models, into a decoder description (XDF). Finally, we take MPEG RVC as the base application for JADE, but the concept of ADM representation can also be extends to others Medias that involve signal processing, such as audio, cryptographic or 3D applications.

References

1. Amer, I., Lucarz, C., Mattavelli, M., Roquier, G., Raulet, M., Déforges, O., Nezan, J.F.: Reconfigurable Video Coding: The Video Coding Standard for Multi-core Platforms. In: IEEE Signal Processing Magazine(to appear), Special Issue on Signal Processing on Platforms with Multiple Cores (2009)
2. Barrett, E.: 3c - a JIT compiler with LLVM. Tech. rep., Bournemouth University (2009). URL http://llvm.org/pubs/2009-05-21-Thesis-Barrett-3c.html
3. Bhattacharya, B., Bhattacharyya, S.: Parameterized dataflow modeling for DSP systems. Signal Processing, IEEE Transactions on **49**(10), 2408–2421 (2001). DOI 10.1109/78.950795
4. Bhattacharyya, S.S., Eker, J., Janneck, J.W., Lucarz, C., Mattavelli, M., Raulet, M.: Overview of the MPEG reconfigurable video coding framework. Journal of Signal Processing Systems (2009)
5. Bhattacharyya, S.S., Murthy, P.K., Lee, E.A.: APGAN and RPMC: Complementary heuristics for translating DSP block diagrams into efficient software implementations. Journal of Design Automation for Embedded Systems. In: DSP Block Diagrams into Efficient Software Implementations, DAES, pp. 33–60 (1997)
6. Bilsen, G., Engels, M., Lauwereins, R., Peperstraete, J.: Cycle-static dataflow. Signal Processing, IEEE Transactions on **44**(2), 397–408 (1996). DOI 10.1109/78.485935
7. C. Lattner and V. Adve: LLVM Language Reference Manual. Tech. Rep. 4th edition, ECMA International (2006). Available at: http://llvm.cs.uiuc.edu/docs/LangRef.html
8. Eker, J., Janneck, J.: CAL Language Report. Tech. Rep. ERL Technical Memo UCB/ERL M03/48, University of California at Berkeley (2003)
9. Gorin, J., Raulet, M., Cheng, Y., Lin, H., Siret, N., Sugimoto, K., Lee, G.: An RVC Dataflow Description of the

AVC Constrained Baseline Profile Decoder. In: Proceedings of ICIP'09 (2009)
10. Gu, R., Janneck, J.W., Bhattacharyya, S.S., Raulet, M., Wipliez, M., Plishker, W.: Exploring the concurrency of an MPEG RVC decoder based on dataflow program analysis. IEEE Transactions on Circuits and Systems for Video Technology (2009)
11. Janneck, J., Miller, I., Parlour, D., Roquier, G., Wipliez, M., Raulet, M.: Synthesizing hardware from dataflow programs: an MPEG-4 Simple Profile decoder case study. In: Signal Processing Systems (SiPS) (2008)
12. kwong Kwok, Y., Rol, P., Chin, T., Department, H.O.: High-performance algorithms for compile-time scheduling of parallel processors. In: 83 - PhD. Thesis, HKUST, Hong Kong, pp. 90–101 (1997)
13. Lattner, C.: LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL (2002). *See* `http://llvm.cs.uiuc.edu`.
14. Lattner, C., Adve, V.: 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 (2004)
15. Lee, E., Messerschmitt, D.: Synchronous data flow. Proceedings of the IEEE **75**(9), 1235–1245 (1987)
16. Lee, E.A., Parks, T.M.: Dataflow Process Networks. Proceedings of the IEEE **83**(5), 773–801 (1995)
17. Mattavelli, M., Amer, I., Raulet, M.: The reconfigurable video coding standard [standards in a nutshell]. Signal Processing Magazine, IEEE **27**(3), 159 –167 (2010). DOI 10.1109/MSP.2010.936032
18. von Platen, C., Eker, J.: Efficient realization of a CAL video decoder on a mobile terminal (position paper). In: Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on, pp. 176–181 (2008). DOI 10.1109/SIPS.2008.4671758
19. Plishker, W., Sane, N., Kiemb, M., Anand, K., Bhattacharyya, S.S.: Functional DIF for rapid prototyping. In: Proceedings of the International Symposium on Rapid System Prototyping, pp. 17–23. Monterey, California (2008)
20. Richardson, I., Bystrom, M., Kannangara, S., Frutos, D.: Dynamic Configuration: Beyond Video Coding Standards. In: IEEE System on Chip Conference. IEEE (2008). URL http://www.openvideocoding.org/
21. Ritz, S., Pankert, M., Zivojinovic, V., Meyr, H.: Optimum vectorization of scalable synchronous dataflow graphs. In: Application-Specific Array Processors, 1993. Proceedings., International Conference on, pp. 285–296 (1993). DOI 10.1109/ASAP.1993.397152
22. Sangappa, S., Palaniappan, K., Tollerton, R.: Benchmarking Java against C/C++ for interactive scientific visualization. In: JGI '02: Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande, p. 236. ACM, New York, NY, USA (2002). DOI 10.1145/583810.583848. URL http://dx.doi.org/10.1145/583810.583848
23. Singer, J.: JVM versus CLR: a comparative study. In: PPPJ '03: Proceedings of the 2nd international conference on Principles and practice of programming in Java, pp. 167–169. Computer Science Press, Inc., New York, NY, USA (2003)
24. Wernli, L.: Design and Implementation of a Code Generator for the CAL Actor Language. Tech. Rep. UCB/ERL M02/5, EECS Department, University of California, Berkeley (2002). URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2002/3965.html
25. Wipliez, M., Raulet, M.: Classification and Transformation of Dynamic Dataflow Programs (in submission). In: submitted to conference on Design and Architectures for Signal and Image Processing (DASIP 2010) (2010)