AN LLVM-BASED DECODER FOR MPEG RECONFIGURABLE VIDEO CODING

J. Gorin[1] *- M. Wipliez*[2] *- J. Piat*[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) 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 dataflow programs described in *RVC-CAL Actor Language* (RVC-CAL) [1]. RVC-CAL naturally expresses potential parallelism between tasks of an application, which makes an ADM description suitable for implementation to a wide variety of platform, from uniprocessor systems to FPGAs. MPEG RVC eases the conception process of decoders by describing decoders at a library component level instead of using monilithic 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) capacities that allow the conception of the first decoder able to dynamically instantiate an RVC decoder description. This decoder, unlike static decoders generated by RVC tools [2], keeps *de facto* the features of an RVC description namely portability, scalability and reconfiguration ability.

Index Terms— Reconfigurable Video Coding, RVC-CAL Actor Language, MPEG decoder, Low Level Virtual Machine, Dataflow programming, Multi-core, Code synthesis

1. INTRODUCTION

MPEG multimedia coding technology has delivered many heterogeneous coding standards. The specification of such standards has been done case-by-case providing monolithic textual description and reference software in C++. Very little attention has been given to provide a formalism that would explicitly present the common components of different standards and that would furthermore be suited to the heterogeneous panel of systems available nowadays. MPEG structured their standards with many possibilities of algorithm combination called *Profiles*. Selecting an appropriate profile for a selected platform would enable a codec designer to have any desired trade-off between compression performance and implementation complexity. Currently, interoperability demands that the selection process for a profile be described into the normative description of a codec, or at best, into a number of choices codified within the media syntax.

MPEG 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. Our work presented in 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 and universal MPEG decoder engine that can configure and reconfigure an MPEG RVC decoder description. This decoder has been successfully tested onto multiple platforms.

2. MPEG RECONFIGURABLE VIDEO CODING

Since the year 2004, MPEG has become very active about video compression/decompression application prototyping coming from their coding standards. These works led to the creation of the MPEG Reconfigurable Video Coding framework. This framework aims at "allowing a dynamic development, implementation and adoption of standardized video coding solutions with features of higher flexibility and reusability" [1]. Its key approach is to provide an *Abstract Decoder Model* (ADM) of existing or new MPEG standards at system-level suited for any platform.

An Abstract Decoder Model, shown in Figure 1, is a generic representation of a decoder, built as a block diagram expressed with the *XML Dataflow Format* (XDF) [3]. XDF is an XML dialect that describes the connections between

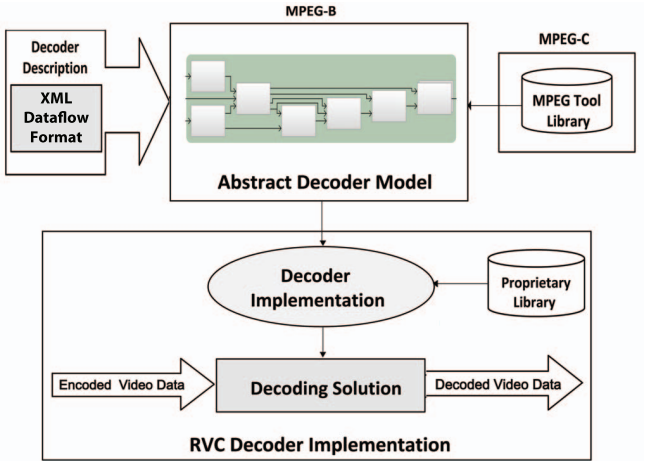

Fig. 1. The reconfigurable Video Coding Framework.

blocks called *Functional Units* (FUs). Each FU defines processing entities of a decoder and connections represent data flow between FUs. MPEG RVC is under development as part of the MPEG-B standard [3], which defines the framework and the language used to describe components; and as part of the MPEG-C [4] standard, which defines the library of video coding tools (*Video Tools Library* or VTL) employed in existing MPEG standards.

MPEG RVC provides both a normative standard library of FUs and a set of decoder descriptions expressed as networks of FUs. An ADM representation of a decoder is modular and helps its reconfiguration by allowing the topology of its network to be easily modified. RVC mainly focuses on reusability by allowing different decoder descriptions to instantiate common FUs across standards.

2.1. RVC-CAL Dataflow Programming

FUs of an Abstract Decoder Model are defined using RVC-CAL, a subset of CAL Actor Language. RVC-CAL [5] is a textual and domain specific language for writing dataflow models, more precisely for defining Functional Units (or *actors*) of a decoder at a high level of description. An actor represents an autonomous entity, thus 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. RVC-CAL, compared with CAL, restricts the data types, and operators that cannot be easily implemented onto the 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

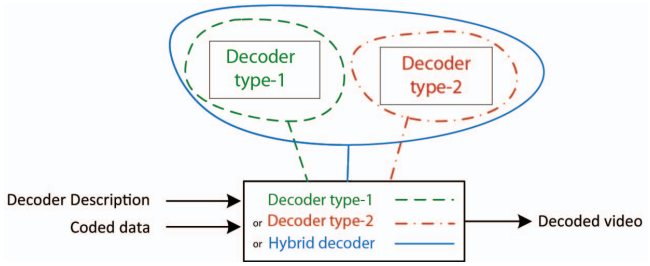

Fig. 2. Representation of an RVC decoder.

produces sequences of tokens to output ports. Actions have data-dependent conditions for their execution. The execution of an action may change the actor internal state, so that the produced output sequences are functions of the consumed input sequences and of the current actor state.

2.2. RVC Decoder features

The main benefit of the MPEG RVC approach is to be able to describe a decoder without carrying implementation details. By using an RVC-CAL description, an Abstract Decoder Model has two benefits:

1. *Parallelism scalability:* While scalability of parallelism composes the principal weakness of traditional imperative languages, RVC-CAL proposes to encapsulate tasks into actors, explicitly exposing parallelism between tasks of an application.

2. *Modularity:* The "untimedness" and the strong encapsulation of actors offer high degree of modularity. This property allows each actor of a dataflow model to be unbound to its environment and thus to be easily used in others.

C and HDL synthesis tools [1] can be used to automatically generate static decoders for specific platforms. These decoders may only take the useful information from an ADM description to fully exploit processing resources of the targeted platform. Our approach differs in the way that we skip this synthesis process and dynamically run an ADM description on the targeted platform. To be fully RVC compliant, this decoder must integrate a Video Tools Library, a library that supports RVC ADM representation and a Virtual Machine able to produce efficient *Just-In-Time* (JIT) compilation. By combining all these tools, video decoders (Fig. 2) can be created, configured and reconfigured adaptively to decode any video sequence or syntax.

3. LOW LEVEL VIRTUAL MACHINE (LLVM)

Performance and portability are crucial to develop a sustainable RVC decoder. High Level Language Virtual Machines

(the most popular are *Java Virtual Machine* or *Common Language Runtime*) have generally high portability but poor performance compared to an equivalent C application [6, 7]. The *Universal Video Decoder* (UVD) [8], dedicated to video decoding, shows interesting performances but at the cost of developing a dedicated Virtual Machine, currently tested on a single environment. 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. LLVM Intermediate Representation

The centerpiece of LLVM is the LLVM *Intermediate Representation* (IR) using LLVM virtual instruction set. The LLVM instruction set is a very simple representation and language independent type-system that captures the key operations of ordinary processors but avoids machine specific constraints such as physical registers, pipelines, low-level calling conventions, or traps. The generic instruction set of LLVM constitutes an excellent low-level representation for FUs, allowing their representation to be directly executed into a wide variety of platforms. LLVM IR includes explicit type information, control flow graphs, dataflow representation along with a *three-address code* (3AC) and *Static Single Assignment* (SSA) form. These properties simplify transformations and code analyses to allow aggressive multistage optimization and high-performance execution onto its integrated Virtual Machine [7] .

3.2. LLVM compilation framework

LLVM also designates a compilation framework that exploits the LLVM IR to provide a combination of features not available in any previous compilation approach[7]. 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.

This 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.

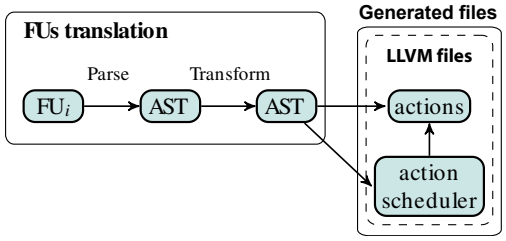

Fig. 3. LLVM Code Generation for a selected *FUi*.

4. PORTABLE VIDEO TOOL LIBRARY GENERATION

The main purpose of this paper is to use the efficiency of the LLVM representation to provide a portable VTL manageable by the LLVM infrastructure. We describe in this part the translation process used to convert FUs described in RVC-CAL into an LLVM representation.

4.1. Functional Units transformation

The first constraint of the LLVM code generation is to avoid lost information from the original VTL, described in RVC-CAL. We used mechanisms presented in [2], which describe the translation of an RVC ADM into a C representation for single processors. This C representation transformed FUs from VTL into several instances. As an instance is equivalent to FU but with resolved parameters, a direct translation of a C representation into LLVM would induce loss information from the VTL. In our approach, we extend the method presented in this previous paper by applying our transformation directly to the VTL and thus conserve modularity properties of FUs.

The first stage of our translation process, described in Figure 3, consists in parsing each Functional Unit from the VTL to an Abstract Syntax Tree (AST). This AST helps the application of a series of transformation and the conversion of the RVC-CAL into LLVM IR. The transformations applied to the AST are:

- *Control Flow Graph* (CFG) transformation: Actions are transformed in a set of basic blocks, and each basic block as a sequence of instructions, ending in exactly one terminator instruction.

- *Static Single Assignment* (SSA) and *Three-Address Code* (3AC) form: Each register is written in exactly

Fig. 4. Orcc framework with available backends.

one instruction, that takes one or two operands and produce a single result.The LLVM instruction set includes an explicit phi instruction, which corresponds directly to the standard (non-gated) ϕ function of SSA form[7].

- *Strong type checking*: Each variable is declared as integer type with an arbitrary width. This transformation checks and ensures the size coherence for each variable contained in an LLVM IR.

As presented in section 2.1, actions from an FU have data-dependent conditions for their execution. We composed an actor with several functions that describe actions and an *action scheduler*. The action scheduler checks the presence and values of tokens on its input ports and executes actions that fit to their associated executing conditions. The rationale for using action scheduler is to conserve the encapsulation of an actor and to easily deport the execution of the global network to a scheduler inside the decoder of the targeted platform; we called it *actor scheduler* and detailed it on part 5.2.

4.2. LLVM code generator into the Open RVC-CAL Compiler framework

To ease the translation process of FUs, we integrate the LLVM actor code generation into the extensible Open RVC-CAL Compiler[1] (Orcc) Infrastructure. This compiling infrastructure, shown in Figure 4, has been structured to easily write code generators that targeting any language from a single Intermediate Representation. The Orcc specific Intermediate Representation preserves the properties of RVC-CAL representation while being at a lower level of description, with simple arithmetic expression, imperative statements and *SSA* property. Back-ends are already available for generating VHDL, Java, C and C++ representations from an Orcc IR.

For the need of our portable VTL, we developed a new back-end that converts the Orcc IR of RVC-CAL Functional Units into an LLVM Representation. As the Orcc IR directly embeds simple arithmetic expressions and SSA form, the translation of an RVC-CAL FU into an LLVM representation is greatly simplified. The developed LLVM backend transforms the Orcc IR from each coding tools of the VTL into a set of LLVM files, representing an only *VTL* portable on any platform supported by the LLVM infrastructure.

MPEG RVC is based on the idea that most of the tools used in video coding come from MPEG Standards. As the LLVM representation of coding tools are generics and directly suited for a wide range of platforms, our approach allows decoders to extend the RVC concept by providing FUs outside the VTL. Thus, a decoder can be equally formed by coding tools standardized or not standardized in MPEG-C, for instance by coding tools directly included inside the bitstream to decode. This opportunity opens significant new opportunities in multimedia coding, especially in adaptative decoding [8].

[1]Orcc is available at: http://orcc.sourceforge.net

5. RVC DECODER ENGINE

The second step of our approach is to produce a decoder that dynamically instantiates an RVC description of a decoder according to a network description and our portable VTL. To achieve this goal, we create an engine called *RVC Decoder Engine*, which manages the LLVM infrastructure for dynamically translating LLVM IR into optimized machine code. This RVC decoder engine also integrates execution rules to use the generated decoder.

5.1. Network instantiation

The instantiation of a network designates the initialization of actors, structures and fifos of an RVC description. The dynamic instantiation is made possible by coupling the *Linker*, *Optimizer* and *Just-In-Time* Compiler from the LLVM Infrastructure with an XDF parser and our RVC decoder engine. The RVC decoder, described on Figure 5, has the role to manage the instantiation of a network in a six steps process:

1. *XDF parsing*: XDF description is parsed into a graph that describes the whole network. This graph contains information about the list of FUs in the VTL, instances of an FU and connections in the network.
2. *FUs selection*: The necessary FUs are taken from the portable VTL and stored in memory in the form of LLVM modules.
3. *Linking*: Functions of FUs are changed to their instance name to avoid ambiguous calls. All functions are then links into a single module.
4. *Optimization*: The generated single module passes through aggressive optimizations, selected according to their efficiency for the targeted platform.
5. *Compilation*: Functions of the single LLVM module representing actions and action schedulers are dynamically translated into machine code.
6. *FIFO instantiation*: FIFOs are instantiated and connected to each functions. Our FIFOs are unidirectional circular buffers that avoid the use of semaphore.

The generated decoder is finally connected to the source that contained the encoded video bitstream.

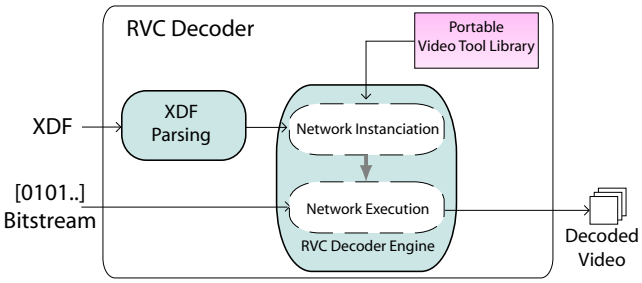

Fig. 5. RVC decoder structure.

5.2. Network execution

Once the decoder is created, the RVC decoder engine executes the whole network. The network scheduler we wrote is a simple round-robin scheduler following the *Dataflow Process Networks* (DPN) [9] model. A DPN is a *model of computation* (MoC) where a group of sequential processes (in our case the *actors*) are communicating through unbounded FIFO channels. DPN models must be dynamically scheduled, hence our *actor scheduler* endlessly calls the *action scheduler* of every actor contained in the network. Actors in the network are scheduled one after the other according to an arbitrary order, in our case the XDF Parsing order.

The strength of our 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 sequence of actions. Therefore, the execution of the network can be made by more than one round robin scheduler. Starting from this observation, a round robin scheduler can contain from one to all the actor of a network. If a platform contains multiple cores, a group of FUs could be mapped onto several round robin schedulers and each scheduler can be affected onto a separate core of the platform. This scheduler enhances parallelism and pipelining of each process for 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 [10].

6. EXPERIMENTAL RESULTS

This section presents the experimental results led on our decoder. These experiments shows that our VTL is portable and compares the achievement of the RVC decoder with an equivalent static C and Java decoders onto several *Operating Systems* (OS). With this aim in mind, we developed our RVC decoder, described in Figure 3, in C++ using LLVM 2.6[2]. We also used our LLVM backend to generate a portable VTL from RVC-CAL FUs standardized in MPEG-C. The FUs currently available in MPEG-C are the coding tools from MPEG-4 Part-2 *Simple Profile* (SP) and MPEG-4 *Advanced Video Coding* (AVC) standards. We compiled our developed RVC decoder and copied our generated portable VTL 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*.

		FUs
	MPEG-4 *Simple Profile*	22
+	MPEG-4 *Advanced Video Coding*	47
	Generated VTL	69

Table 1. Description of the generated portable VTL.

Table 1 shows the number of FUs contained in the VTL and currently available in MPEG-C. Those two configurations of decoders described in [1] for the MPEG-4 Part-2 *Simple Profile* (SP) decoder and from [11] for the MPEG-4 *Advanced Video Coding*(AVC) *Constrained Baseline Profile* (CBP) decoder are the most representative 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.

	C	JAVA	LLVM
Windows (Visual Studio)	26,7 fps	3,5 fps	24,9 fps
Linux (GCC)	39,3 fps	4,83 fps	24,6 fps
MAC OS X (Xcode)	26,6 fps	4,85 fps	26,4 fps

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

The same configuration of these two decoder has also been generated by the static C and Java synthesis tools included in RVC framework. The comparison results, showed in Table 2 and Table 3, compare the achievement of our RVC decoder with these two static implementation. These decoders were respectively tested on conformance testing sequences for SP decoders[3] and AVC decoders[4].

A comparison of our decoder implementation with the equivalent static C decoders (i.e. without any Virtual Machine) shows that the impact of the LLVM Virtual Machine is unseen on Windows and Mac (OS) and has minor impact (roughly 20%) on Linux OS. On the contrary, our dynamic

[2]LLVM 2.6 is available at: http://llvm.org/
[3]Video sequences available at: http://standards.iso.org/
[4]Video sequences available at: http://wftp3.itu.int/av-arch/jvt-site/

decoder is about 7 times faster than a static Java decoders running on the *Java Virtual Machine* (JVM). This speed factor can be explained by the fact that Java has no pointer mean. Each access to a FIFO in Java involves a memory copy of data, which puts pressure on the garbage collector of the Virtual Machine.

	C	JAVA	LLVM
Windows (Visual Studio)	30,9 fps	2,8 fps	31,7 fps
Linux (GCC)	48,5 fps	2,35 fps	33,5 fps
MAC OS X (Xcode)	34,9 fps	2,78 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 ms and about 1 second for the MPEG-4 AVC description. The reconfiguration times can be greatly improved by encapsulating dataflow network representation into bitstream and by supporting partial reconfiguration of decoders. We also tested the multi-core abilities of our decoder for a dual core processor. Two POSIX threads were used 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 performance shows benefits of the multi-core (up to 1.5) depending on the configuration used. We have to find some mechanisms to have a smart and automatic dispatchment of the actors into several schedulers.

7. CONCLUSION AND PERSPECTIVES

The concept of such an RVC decoder implementation opens significant new opportunities, first in video decoding then in sound, 3D or any other medium that involve the purpose of a decoder. The main advantage of our approach is to couple the active research of LLVM and MPEG RVC. Its implementation, integrable into a wide variety of platforms, represents a good starting point for new research on adaptive decoding. The experimental results show the relevance of our approach by testing the portability, scalability and the fast configuration of two decoder samples.

However, this decoder is a base for many challenges. The DPN model of computation we use can be improved with dataflow analysis to be close to an equivalent sequential program. We plan to reduce the overhead induced by our scheduler by finding some mechanisms that could predict the next actions/actors to execute. The reconfiguration of a decoder involves yet a global change in the new decoder. We also plan to reduce reconfiguration times, by finding parts of a decoder that really needs to be changed. Finally, we prove that this decoder has scalable parallelism and rules should been implemented to automatically dispatch decoding algorithms onto the processing elements of a platform. The abilities brought by our modular and promising approach still require many new researches to produce real time High-Definition decoders.

8. REFERENCES

[1] 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.

[2] Matthieu Wipliez, Ghislain Roquier, and Jean-François Nezan, "Software Code Generation for the RVC-CAL Language," *Journal of Signal Processing Systems*, 2009.

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

[4] ISO/IEC FDIS 23002-4: 2009, "Information technology - MPEG video technologies - Part 4: Video Tool Library," 2008.

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

[6] Jeremy Singer, "JVM versus CLR: a comparative study," in *Proceedings of the 2nd international conference on Principles and practice of programming in Java (PPPJ'03)*, New York, USA, 2003, Computer Science Press, Inc.

[7] Chris Lattner and Vikram 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.

[8] I. Richardson, M. Bystrom, S. Kannangara, and De Frutos, "Dynamic Configuration: Beyond Video Coding Standards," in *IEEE System on Chip Conference*. IEEE, September 2008.

[9] Edward A. Lee and Thomas M. Parks, "Dataflow Process Networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, May 1995.

[10] 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*, November 2009.

[11] 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.