Synthesizing hardware from dataflow programs: An MPEG-4 simple profile decoder case study

Jörn W. Janneck, Ian D. Miller, David B. Parlour, Ghislain Roquier,

Matthieu Wipliez, Mickael Raulet

▶ To cite this version:

HAL Id: hal-00336518

https://hal.archives-ouvertes.fr/hal-00336518

Submitted on 4 Nov 2008

SYNTHESIZING HARDWARE FROM DATAFLOW PROGRAMS: AN MPEG-4 SIMPLE PROFILE DECODER CASE STUDY

Jörn W. Janneck[1], Ian D. Miller[1], David B. Parlour[1],
Ghislain Roquier[2], Matthieu Wipliez[2], Mickaël Raulet[2]

[1]Xilinx Inc, San Jose, CA 95123, U.S.A.
[2]IETR/INSA. UMR CNRS 6164, F-35043 Rennes, France

ABSTRACT

The MPEG Reconfigurable Video Coding working group is developing a new library-based process for building the reference codecs of future MPEG standards, which is based on dataflow and uses an actor language called CAL. The paper presents a code generator producing RTL targeting FPGAs for CAL, outlines its structure, and demonstrates its performance on an MPEG-4 Simple Profile decoder. The resulting implementation is smaller and faster than a comparable RTL reference design, and the second half of the paper discusses some of the reasons for this counter-intuitive result.

Index Terms— Dataflow, CAL, Reconfigurable Video Coding, MPEG, high-level synthesis

1. INTRODUCTION

The increasing complexity of video codecs has made it increasingly difficult to accompany video standards with reliable reference implementations built from scratch. For this reason, MPEG has decided to explore a library-based approach—a library of video coding modules defines the basic capabilities of a standard, and future standards, rather than building an independent new library, extend the existing code base with new functionality. MPEG's *Reconfigurable Video Coding* effort is in the process of constructing and standardizing this library.

In addition to the library itself, RVC is also concerned with the language for describing individual modules, and the way in which they are composed into working decoders. Adopting dataflow as the fundamental design methodology, they have decided to use the CAL actor language [1] for building the modules, which are composed using an XML format called *DDL* [2, 3].

The use of dataflow as a specification language for video codecs opens interesting new opportunities. In the past, the reference code was at best a starting point for actual implementations. Especially hardware implementations could not directly be derived from the sequential software that served as an executable reference. Dataflow programs, on the other hand, are naturally concurrent, and a much better starting point for a range of efficient implementations, from sequential software, to multi-core architectures, to programmable hardware to ASICs.

This paper presents a tool that translates dataflow programs written in CAL into RTL descriptions suitable for implementation in programmable hardware, and its application to the construction of an MPEG-4 Simple Profile decoder. After reviewing the dataflow programming model and some basic properties of the CAL actor language in section 2 and the tools supporting MPEG's RVC effort in section 3, we present the MPEG-4 decoder design and its translation to hardware in section 4, explaining the various stages in the translation process. The quality of the resulting decoder implementation turns out to be better than that of a VHDL reference design, and section 5 discusses how some aspects of the dataflow design process contribute to this surprising result. Finally, section 6 closes with a discussion of the work and some conclusions.

2. DATAFLOW AND CAL

The *dataflow* programming model presented in this paper composes systems from computational kernels called *actors* by connecting them using lossless directed FIFO channels. Through these channels they send each other packets of data, called *tokens*. Depending on the implementation platform, the FIFOs may be bounded or unbounded, and size constraints may or may not apply to individual tokens. Our model is a slight generalization of the one presented in [4], permitting actors that are non-deterministic and not prefix-monotonic.

Actors themselves are written in the CAL actor language [1]. A detailed discussion of CAL is beyond the scope of this paper, but for our purposes it is sufficient to say that it provides the syntactical constructs to specify the essential pieces of an actor, viz. its input ports and output ports, a definition of the variables containing its internal state, and a number of transition rules called *actions*. Each actor executes by making discrete, atomic *steps* or *transitions*. In each step, it picks exactly one action from its set of actions (according to the conditions associated with that action), and then executes

it, which consists of a combination of the following things:

1. consume input tokens,

2. produce output tokens,

3. modify the state of the actor.

The state of an actor is strictly local, i.e. it is not visible to any other actor. The absence of shared state is what allows the actors in a system to execute their actions without being concerned about race conditions on their state.

Actors are similar to objects in object-oriented programming in the sense that they encapsulate some state and associate it with the code manipulating it (the actions). They differ from objects in that actors cannot *call* each other—there is no transfer of control from one actor to another, each actor can be thought of as its own independent thread.

3. DATAFLOW TOOLS FOR RVC

3.1. Simulator

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 actors 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). Contrarily to Moses, this project does not provide a network graphical editor. Networks have been traditionally described in a textual language called Network Language (NL), which can be automatically converted to DDL and vice versa. It is also possible to use the Graphiti editor[3] to display networks in the DDL format.

3.2. Hardware synthesis

The work presented here is an available tool that converts CAL to HDL. After parsing, CAL actors are instantiated with the actual values for their formal parameters. The result is an XML representation of the actor which is then precompiled (transformation and analysis steps, including constant propagation, type inference and type checking, analysis of data flow through variables...), represented as a sequential program in static single assignment (SSA) form (making explicit the data dependencies between parts of the program).

Then follows the synthesis stage, which turns the SSA threads into a web of circuits built from a set of basic operators (arithmetic, logic, flow control, memory accesses and the like). The synthesis stage can also be given directives driving

the unrolling of loops, or the insertion of registers to improve the maximal clock rate of the generated circuit.

The final result is a Verilog file containing the circuit implementing the actor, and exposing asynchronous handshake-style interfaces for each of its ports. These can be connected either back-to-back or using FIFO buffers into complete systems. The FIFO buffers can be synchronous or asynchronous, making it easy to support multi-clock-domain dataflow designs.

3.3. Software synthesis

It is important to be able to automatically obtain a concrete software implementation from a dataflow description. The C language is particularly well-suited as a target language. The same code can be compiled on any processor, from embedded DSPs and ARMs to general-purpose microprocessors, which considerably eases the task of writing a software synthesis tool. The interest of having an automatic C software synthesis is two-folded. The code obtained can be executed, in which case it enables a considerably faster simulation of the dataflow program and the ability to debug the program using existing IDEs (Visual Studio, Eclipse CDT). The C code description may be a basis for a tailor-made decoder. For these reasons, we created the Cal2C tool [5] that aims at producing functionally-equivalent, humanly-readable C code from CAL descriptions.

The Cal2C compilation process has been successfully applied to the MPEG-4 Simple Profile dataflow program written by the MPEG RVC experts (Fig. 1). The synthesized model is compared to CAL dataflow program simulated with the Open Dataflow environment so as to validate the Cal2C tool. The synthesized software is faster than the CAL dataflow simulated (20 frames/s instead of 0.15 frames/s), and close to real-time for a QCIF format (25 frames/s). It is interesting to note that the model is scalable: the number of macroblocks decoded per second remains constant when dealing with larger image sizes. Using Cal2C has also permitted to correct some actors which had an implementation-dependent behavior, such as the assumption of a particular action schedule.

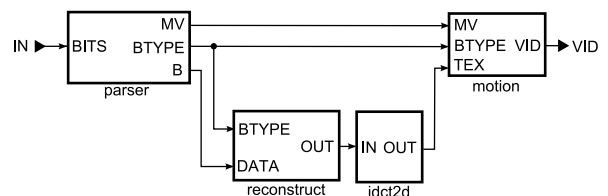
Fig. 1. Top-level view of the MPEG decoder, depicting parser, AC/DC reconstruction, IDCT, and motion compensation.

---

[1]http://www.tik.ee.ethz.ch/ moses/
[2]http://opendf.sourceforge.net/
[3]http://sourceforge.net/projects/graphiti-editor

4. SYNTHESIZING AN MPEG-4 SP DECODER

The MPEG-4 Simple Profile decoder discussed in this work is a computational engine consuming a stream of bits on its input (the MPEG bitstream), and producing video data on its output. At 30 frames of 1080p per second, this amounts to $30 * 1920 * 1080 =$ approx. 62.2 million pixels per second. In the common YUV420 format, each pixel requires 1.5 bytes on average, which means the decoder has to produce approx. 93.3 million bytes of video data (*samples*) per second.

Fig. 1 shows a top-level view of the dataflow program describing the decoder.[4] The main functional blocks include a parser, an reconstruction block, a 2-D inverse discrete cosine transform (IDCT) block, and a motion compensator. All of these large functional units are themselves hierarchical compositions of actors—the entire decoder comprises of about 60 basic actors.

The parser analyzes the incoming bitstream and extracts the data from it that it feeds into the rest of the decoder. It is by far the most complex block of the decoder, more than a third of the code is used to build the parser. The reconstruction block performs some decoding that exploits the correlation of pixels in neighboring blocks. The IDCT, even though it is the locus of most of the computation performed by the decoder, is structurally rather regular and straightforward compared to the other main functional components. Finally, the task of the motion compensator is to selectively add the blocks issuing from the IDCT to blocks taken from the previous frame. Consequently, the motion compensator needs to store the entire previous frame of video data, which it needs to address into with a certain degree of random access. This data storage and movement results in a few interesting design challenges, some of which are discussed in section 5.

4.1. Hardware synthesis

When generating hardware implementations from networks of CAL actors (specified in some format, such as DDL mentioned above), we currently translate each actor separately, and connect the resulting RTL descriptions with FIFOs. Consequently, we currently do not employ any cross-actor optimizations.

Actors interact with FIFOs using a handshake protocol, which allows them to sense when a token is available or when a FIFO is full. We also do not synthesize any schedule between actors, which means that the resulting system is entirely self-scheduling based on the flow of tokens through it.

The translation of each CAL actor into a hardware description follows a three-step process:[5]

1. instantiation

---

[4]The decoder discussed in this paper is publicly available at http://opendf.sf.net.

[5]The code for instantiation and precompilation is available on http://opendf.sf.net.

2. precompilation
3. RTL code generation

This is followed by the synthesis of the network that connects the actor instances.

Instantiation. The elaboration of the network structure yields a number of actor *instances*, which are references to CAL actor descriptions along with actual values for its formal parameters. From this, instantiation computes a *closed* actor description, i.e. one without parameters, by moving the parameters along with the corresponding actual values into the actor as local (constant) declarations. It then performs constant propagation on the result.

Precompilation. After some simple actor *canonicalization*, in which several features of the language are translated into simpler forms, precompilation performs some basic source code transformations to make the actor more amenable to hardware implementation, such as e.g. inlining procedure and function calls. Then the canonical, closed actors are translated into a collection of communicating threads.

In the current implementation, an actor with N actions is translated into $N + 1$ threads, one for each action and another one for the *action scheduler*. The action scheduler is the mechanism that determines which action to fire next, based on the availability of tokens, the guard expression of each action (if present), the finite state machine schedule, and action priorities.

To facilitate backend processing for both hardware and software code generation, the threads are represented in static single-assignment (SSA) form. They interact with the environment of the actor through asynchronous token-based FIFO channels. Their internal communication happens through synchronous unbuffered signals (this is, for instance, how the scheduler triggers actions to fire, and how actions report completion), and they also have shared access to the state variables of the actor.

RTL code generation. The next phase of the translation process generates an RTL implementation (in Verilog) from a set of threads in SSA form. The first step simply substitutes operators in expressions for hardware operators, creates the hardware structures required to implement the control flow elements (loops, if-then-else statements), and also generates the appropriate muxing/demuxing logic for variable accesses, including the Φ elements in the SSA form.

The resulting basic circuit is then optimized in a sequence of steps.

1. **Bit-accurate constant propagation.** This step eliminates constant or redundant bits throughout the circuit, along with all wires transmitting them. Any part of the circuit that does not contribute to the result will also be removed, which roughly corresponds to dead code elimination in traditional software compilation.

2. **Static scheduling of operators.** By default, operators and control elements interact using a protocol of ex-

plicit activation—e.g., a multiplier will get triggered by explicit signals signifying that both its operands are available, and will in turn emit such a signal to downstream operators once it has completed multiplication. In many cases, operators with known execution times can be scheduled statically, thus removing the need for explicit activation and the associated control logic. In case operands arrive with constant time difference, a fixed small number of registers can be inserted into the path of the operand that arrives earlier.

3. **Memory access optimizations.** Arrays are mapped to RAM primitives for FPGA implementation. Typical FPGA RAM resources range in size from 16 bits (lookup table memory) to 18 kBit or more (block RAM). RAM primitives can be ganged up to form larger memories, or a number of small arrays may be placed into one RAM. Furthermore, these RAM primitives usually provide two or more ports, which allows for concurrent accesses to the same memory region. Based on an analysis of the sizes of arrays and the access patterns, the backend maps array variables to RAM primitives, and accesses to specific ports.

4. **Pipelining, retiming.** In order to achieve a desired clock rate, it may be necessary to add registers to the generated circuit in order to break combinatorial paths, and to give synthesis backends more opportunity for retiming.

Network synthesis. The RTL implementations of all the actors in the system are connected by a network that is obtained from a straightforward translation of the original graph structure into HDL, replacing every dataflow connection with the appropriate handshaking signals that mediate the token traffic between actors. Also, during this step the FIFO buffers are instantiated, sized according to the annotations the user provided in the network description.

The network description also allows the user to add annotations that declare actors to be running in different clock domains. Network synthesis will recognize those and generate the appropriate clock network. It will also use either synchronous or asynchronous FIFO implementations depending on whether the two actors connected by the FIFO are in the same or in different clock domains.

	Size	Speed	Code size	Time
	slices, BRAM	kMB/S	kLOC	MM
CAL	3872, 22	290	4	3
VHDL	4637, 26	180	15	12

The above table shows the quality of the result produced by the RTL synthesis engine for the MPEG decoder. Note that the code generated from the high-level dataflow description actually outperforms the VHDL design in terms of both throughput and silicon area—and this in spite of the fact that our synthesis engine does not perform a number of potential optimizations, such as those involving more than one actor. The next section illustrates some of the reasons for this result.
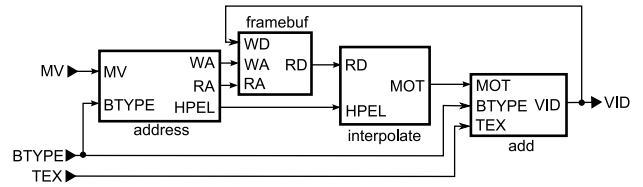

Fig. 2. Basic motion compensator in an MPEG decoder.

5. DATAFLOW DEVELOPMENT PROCESS

In order to illustrate some of the practical aspects of dataflow development, consider the motion compensator subsystem in Fig. 2. Motion vectors come in on its `MV` input port, and from those the `address` actor generates addresses into the frame buffer (in the `framebuf` actor), which retrieves the data, and `interpolate` and `add` proceed to build the final video data.

The engineering challenge is the result of a number of technical constraints. First, we need to produce no fewer than 93.3 million bytes of video data per second in order to do 1080p at 30 frame per second. Furthermore, say we aim for a target clock rate of about 120 MHz. Together with the requirement of 93.3 million samples per second for 1080p at 30 frames per second, this leaves on average no more than 1.29 cycles for each sample. Finally, let us assume we have DRAM with a setup time of 10 cycles, followed by bursts of 4 bytes per cycle.[6]

Motion compensation happens on 8x8 blocks, separately for the Y, U, and V components. For each of those blocks, we need to read a 9x9 block from the frame buffer (because of potential half-pixel interpolation). In a straightforward line-by-line frame buffer organization, we thus need to read, for each block of 8x8=64 samples, 9 bursts of length 3 words (each of those being 4 bytes). This takes at least $9*(10+3) = 117$ cycles—with 64 samples produced, this comes to 1.83 cycles per sample, which is too much. In order to meet the requirement of 1.29 cycles per sample, we need to exploit the locality of the motion compensation, i.e. the fact that adjacent blocks tend to refer to similar regions in the previous frame.

5.1. Caching

One approach might be to use a cache in front of the frame buffer, which reads data from the previous frame in larger bursts and stores them in local memory. The cached image data can be retrieved at a higher rate and single-cycle latency,

---

[6]To simplify the discussion, we assume the DRAM is dual-ported, so that reading and writing to it can be treated independently.
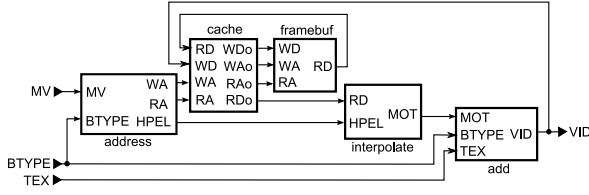
Fig. 3. Motion compensator with caching.

thereby reducing the impact of the setup latency of the frame buffer memory. Fig. 3a shows the modified motion compensator design, with the cache inserted in front of the frame buffer.

In order to test this design, we insert the test cache in Fig. 3. This small actor is parametric in the cache size (number and length of cache lines), logs cache accesses and records the percentage of misses for a given configuration. As a typical result, for 16 lines of 32 samples, we thus obtain a miss rate of 8.3%. Because of the relatively regular access patterns, that rate does not go down significantly by increasing the cache size.

Unfortunately, each of these 8.3% cache misses incurs an 18 cycle penalty (10 cycles setup, 8 cycles per 32-sample burst). Even if the remaining 91.7% of cache hits were infinitely fast, this amounts to $18 * .083 = 1.49$ cycles per sample—better than without the cache, but still not good enough.
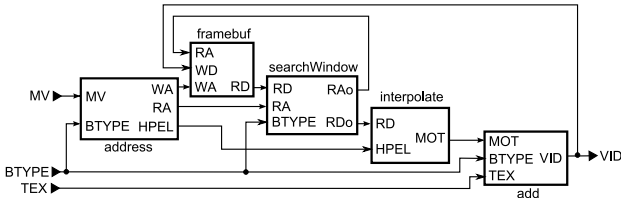

Fig. 4. Motion compensator with prefetch block.

5.2. Prefetch

Besides statistical locality, motion compensation has two other properties that we can exploit: (1) It is always limited to a relatively small search window, and (2) it happens in a predictable and fixed order as we scan down the video frame block by block. In addition, the search windows for neighboring frames mostly overlap, which means that as the decoder advances from one block to the next, the search window shifts some blocks from the previous frame out, and some new ones in, while most data remains.

In our case, the search window is 3x3 macroblocks, where each macroblock consists of 6 blocks of 8x8 samples, or 384 samples. Reading a macroblock from the frame buffer takes

$10 + (384/4) = 106$ cycles, thus reading three macroblocks takes 318 cycles.[7] This has to be done once per macroblock of data, or once every 384 samples—consequently, we spend $318/348 = 0.83$ cycles per sample reading data from the frame buffer, which conveniently meets our requirements. Note that reading the next set of macroblocks can be concurrent with the processing related to the current search window.

5.3. Summary

The design narrative above is intended to illustrate two important aspects of building systems such as the MPEG decoder as dataflow programs. First, the analysis of the cache approach, and its subsequent rejection, happened without ever synthesizing the system to hardware, purely by interactive and quick untimed simulation. In this way we obtained quantitative data (the cache miss rate), which together with some of the engineering constraints (target clock, target sample rate) led us to reject the cache approach. Using high-level tools enabled us to quickly experiment with, and falsify, a design idea without long development cycles, or tedious analyses. The result of this as far as the development process is concerned, is that the dataflow design undergoes many more design cycles than the RTL design—in spite of being done in a quarter of the time. Most of the time in RTL design is spent on getting the system to work correctly. By contrast, a much higher proportion of the dataflow design time is spent on optimizing system performance. At least in the current case study, the positive effects of the shorter design cycles seem to outweigh the inefficiencies introduced through high-level synthesis, and the reduced control of the designer over specific implementation details.

Second, comparing the original design of the motion compensator in Fig. 2 with the design incorporating the prefetched search window in Fig. 4, the difference is exactly one actor, and a few slightly altered connections. None of the other actors in the motion compensator, and of course none of those in the rest of the decoder, were ever touched or modified by this design change. The asynchrony of the programming model, and its realization in hardware, assured that the rest of the system would continue to work in spite of the significantly modified temporal behavior. Any design methodology relying on a precise specification of the timing aspects of the computation—such as RTL, where designers specify behavior cycle-by-cycle—would have resulted in changes rippling throughout the design.

6. DISCUSSION AND CONCLUSION

The central points of this paper can be summarized as follows:

---

[7]This assumes that the frame buffer is structured by macroblocks, so that each macroblock can be read in one burst. This organization would not have made much sense previously, but with the complete prefetched search window in local storage, it can be arranged in this manner.

1. We presented a tool that translates dataflow programs like those in the MPEG RVC framework into efficient implementations in programmable hardware.

2. The high-level design methodology based on dataflow and the CAL actor language has been shown to rival RTL design in terms of the implementation quality, at least in the case we have studied here.

3. We have attributed the surprising quality of the resulting implementation to properties of the dataflow design process, rather than to, e.g., the quality of the translation tool or any particularly sophisticated set of optimizations.

The first point may change the role of the MPEG reference code in future implementation flows—especially when building designs on parallel machines or hardware, dataflow-based reference code is a much better starting point, and the existence of an efficient translation to the target platform means that future video codec implementations may be closer to gradual refinement and optimization of the reference code, rather than from-scratch redesigns of the same functionality. This could drastically improve the productivity of designers in this space, as well as the quality of the resulting designs.

The other two points may be surprising to RTL designers, so they merit closer inspection. Fundamentally, the use of low-level tools seems to create an *illusion of optimality* simply because of the range and detail of control these tools provide to the designer: if we can control every aspect of the design, how could the result be anything but optimal? The answer seems to be that for sufficiently complex designs, there are in fact *too many* things that can be controlled. As a result, a real-world designer with limited time and limited resources will introduce abstractions to make the task intellectually manageable (and, if a group of designers is involved, modularizable), effectively waiving some control in favor of design efficiency.

High-level tools do the same—however, their abstractions are pervasive, consistent, enforced and checked by the tools, and often presented in the form of languages that make it difficult or impossible to break the abstractions. The ad hoc abstractions created by designers may be geared to the specific requirements of an application, but they lack all of the other benefits provided by high-level tools, and often exist only as more or less informal conventions.

The key benefits of the dataflow methodology presented in this paper are the fast design cycles (mostly through eliminating hardware synthesis from the cycle by providing a high-level simulation capability), and a model of strongly encapsulated asynchronously communicating components. Fast design cycles provide a lot of feedback for the designer and frequent opportunity for debugging and performance tuning. The design gets functional sooner, and more time can be spent on optimization.

The dataflow model of strongly encapsulated components that communicate asynchronously has a number of bene-fits for building complex concurrent systems. In this paper we demonstrated one: asynchronous communication makes components naturally less sensitive to the timing properties of their environment, and consequently changes in those properties are less likely to ripple through the rest of the system. A related aspect of the dataflow actor component model is that actors are very flexible with respect to their implementation. For instance, the parser might not need to run at the same speed as the rest of the decoder, as it has much less data to process. Therefore we might consider implementing it in software on a processor. As long as the overall throughput remains sufficient, we can be confident that this choice will not affect the functional correctness of the decoder; the thin FIFO-style interfaces through which actors communicate provide complete abstraction from the specific implementation of an actor.

The work presented in this paper is only the starting point for many potential directions of research. As we have pointed out, the implementation tools themselves provide many opportunities for improvement, involving sophisticated analyses, static scheduling of those parts of a system that can be statically scheduled, cross-actor optimizations, folding, and other program transformations and refactorings such as automatic multi-channelization (i.e. multiplexing the same design for multiple streams of data). Constructing efficient software code generation for CAL (such as the one in [5]), and combining it with the RTL generation to build families of hardware/software systems from one common source is another direction of work, as is the construction of backends that translate dataflow programs to other parallel platforms, such as multi-core architectures.

7. REFERENCES

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

[2] J. Thomas-Kerr, J. W. Janneck, M. Mattavelli, I. Burnett, and C. Ritz, "Reconfigurable Media Coding: Self-describing multimedia bitstreams," in *Proceedings of SIPS'07*, Oct. 2007.

[3] C. Lucarz, M. Mattavelli, J. Thomas-Kerr, and J. W. Janneck, "Reconfigurable Media Coding: a new specification model for multimedia coders," in *Proceedings of SIPS'07*, Oct. 2007.

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

[5] G. Roquier, M. Wipliez, M. Raulet, J. Janneck, I. Miller, and D. Parlour, "Automatic software synthesis of dataflow program: an MPEG-4 Simple Profile decoder case study," in *Proceedings of SiPS'08*, 2008.