Embedded Multi-Core Systems Dedicated to Dynamic Dataflow Programs

Hervé Yviquel, Alexandre Sanchez, Pekka Jääskeläinen, Jarmo Takala, Mickaël Raulet, Emmanuel Casseau

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

Embedded multi-core systems dedicated to dynamic dataflow programs

Hervé Yviquel · Alexandre Sanchez · Pekka Jääskeläinen · Jarmo Takala · Mickaël Raulet · Emmanuel Casseau

Received: date / Accepted: date

Abstract Multimedia applications and embedded platforms are both becoming very complex in order to improve user experience. Thus, multimedia developers need high-level methods to automate time-consuming and error-prone tasks. Dynamic dataflow modeling is attractive to describe complex applications, such as video codecs, at a high level of abstraction. This paper presents a dataflow-based design approach to implement video codecs on embedded multi-core platforms. First, we introduce a custom architecture model to design low-power multi-core chips based on distributed memory and Transport-Triggered Architecture processor cores. Then, we describe software synthesis techniques to improve dynamic dataflow implementations. This methodology has been implemented into open-source tools and demonstrated on video decoders based on the MPEG-4 Visual standard and the new High Efficiency Video Coding standard. The simulations achieve real-time decoding (40FPS) of high definition (720P) MPEG-4 Visual video sequences on a custom multi-core platform clocked at 1Ghz, which is an improvement of more than 100% over previously proposed implementations.

We would like to thank the organizations which have partially funded this work such as the Center for International Mobility (CIMO) and the Academy of Finland (funding decision 253087). We would also give special thanks to the Orcc and TCE communities as a whole for actively participating in the development of the tools which offers solid basements to this work.

Hervé Yviquel, Alexandre Sanchez, Mickaël Raulet
INSA of Rennes, IETR, France.
E-mail: firstname.name@insa-rennes.fr

Pekka Jääskeläinen, Jarmo Takala
Tampere University of Technology, Finland.
E-mail: firstname.name@tut.fi

Emmanuel Casseau
University of Rennes 1, IRISA, Inria, France.
E-mail: emmanuel.casseau@irisa.fr

1 Introduction

Until recent years, the design of the next generation of embedded systems was achieved by increasing chip frequency. But, as for general-purpose computers, embedded systems have hit the *power wall* of the semiconductor technology, forcing chip manufacturers to look towards multi-core architectures to improve the overall system performance. As a result, embedded systems integrate more and more programmable processors, but contrary to general-purpose computers, most of embedded systems are tailored to specific tasks in order to bridge the gap between hardware efficiency and software flexibility.

In parallel, the increasing complexity of data-intensive applications, such as video codecs, along with the emergence of massively parallel architectures, has revived the interest in dataflow programming. Indeed, dataflow programming offers a flexible development approach which is able to build complex and modular applications while modeling parallelism and communication. The efficiency of traditional language programs being the result of 50 years of work on compilers to mainly exploit memory locality, abandoning memory-oriented programming in favor of dataflow programming requires the development of new compilation techniques to fully benefit from the processor architecture.

In this work, we study the modeling and the implementation of data-intensive embedded systems that benefit from dataflow modeling so as to achieve performance constraints imposed by the embedded market. For instance, video decoders have to provide real-time

frame-rates for high-definition video sequences. This paper makes the following contributions:

- We introduce an architecture model dedicated to dynamic dataflow programs that allows design-space exploration of custom embedded multi-core platforms. This architecture model is based on distributed memory organization and exposed-datapath core architecture so as to improve the global efficiency of the platform (power consumption and decoding frame-rate).
- We present a set of advanced software synthesis techniques, based on preliminary work [34], that enhance the performance of implementations of dynamic dataflow programs using their specific properties and the flexibility of software systems over hardware systems.

Our design approach has been implemented into open-source tools and demonstrated on well-known video decoders, including one based on the new High Efficiency Video Coding (HEVC) standard. Using FPGA prototyping and instruction-set simulation, we have evaluated the current top-level performance bound of their implementations on a set of multi-core platforms that target Integrated Circuit implementation.

The paper is organized as follows. First, the specific application model which supports our design approach is described in Section 2. Then, we introduce in Section 3 the architecture model that has been defined specifically for the application model. Next, we describe in Section 4 our software synthesis methodology to implement dynamic dataflow programs on multi-core platforms based on our architecture model. Section 5 presents experimental results and deeply analyzes our implementations of video decoders. Finally, we conclude in Section 7.

2 Application model

Our methodology relies on a programming model based on the dataflow principle [19, 20]. Indeed, dataflow programming offers a flexible development approach which is able to build modular applications while expressing parallelism and communication explicitly. Thus, dataflow programming is very attractive to implement data-intensive applications on embedded multi-core platforms.

2.1 Dataflow modeling

A Model of Computation (MoC) is an abstract specification of how a computation can progress. A MoC is useful to define the semantics of a programming model, i.e. the type of components it can contain and the way they interact.

Existing dataflow MoCs can be split into two main classes: The *static* MoCs [19] allow a predictable behavior such as the scheduling can be done at compile time, in other words statically-defined production/consumption rates. The *dynamic* MoCs allow a data-dependent behavior [20]. Paradoxically, most of the studies stay focused on static dataflow programming [26, 2], even if the development process of complex applications such as video codecs is largely simplified by the expressiveness and the practicality offered by dynamic dataflow programming. Indeed, modern video decoders support advanced features that require a certain expressiveness. For example, the frames of a video sequence can be decomposed in pixel blocks of different sizes (like the Coding-Tree Unit or the tiles of HEVC).

The need for a trade-off between expressiveness and predictability has brought the definition of so-called "quasi-static" dataflow models [5, 10, 3]. Quasi-static dataflow differs from dynamic dataflow in that there are techniques that statically schedule as many operations as possible so that only data-dependent operations are scheduled at runtime. However, even if they seem promising, quasi-static models are not yet mature enough. To our knowledge, quasi-static dataflow-based implementations of complex applications, such as video codecs, have not yet been demonstrated.

2.2 Dynamic dataflow programming

Dynamic dataflow programs rely upon a MoC called Dataflow Process Network (DPN) [20], which is closely related to Kahn Process Network (KPN) [16]. In this model, an application is represented as a directed graph $G = (V, E)$, see Figure 1, such that V is a set of vertices that represent computational units, called *actors*, and E is a set of unidirectional edges that represent unbounded communication channels based on FIFO principle. A FIFO channel $e \in E$ can be empty, denoted as $\perp$, or can carry a possibly infinite sequence of data $X = [x_1, x_2, ...]$ wherein $x_i \in X$ are atomic data called tokens.

Additionally to the KPN model, DPN introduces the notion of firing. An actor firing is an indivisible quantum of computation which corresponds to a mapping function $f \in F$, called action, of input tokens to output tokens applied repeatedly and sequentially on one or more data streams. This mapping is composed of three ordered and indivisible steps: data reading, then computational procedure, and finally data writing. These functions are guarded by a set of firing rules R which specifies when the functions can be fired, i.e.

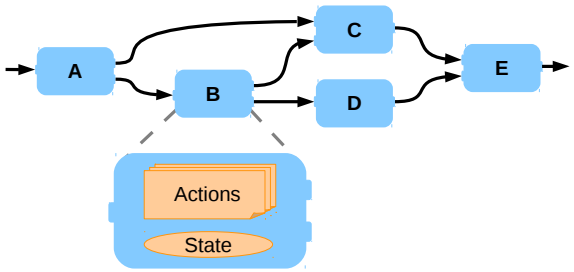


Fig. 1: A simple network wherein the actors contain their own state, actions and firing rules

the number and the values of tokens that have to be available on the input ports to fire the actor. More formally, every actor $a \in V$ is associated with its own set of firing function F_a, and firing rules R_a such that $F_a = [f_1, f_2, ..., f_M]$ and $R_a = [\mathbf{R}_1, \mathbf{R}_2, ..., \mathbf{R}_N]$ within each function $f_i \in F_a$ is associated to a given firing rule $R_i \in R_a$.

A firing rule $\mathbf{R}_i$ defines a finite sequence of patterns, one for each input m of the actor such as $\mathbf{R}_i = [P_{i,1}, P_{i,2}, ..., P_{i,m}] \in S^m$. A pattern $P_{i,j}$ is an acceptable sequence of tokens in R_i on one input j from the input m of an actor. It is satisfied if and only if $P_{i,j} \sqsubseteq X_j$ where X_j is the sequence of tokens available on the j^{th} FIFO channel. The pattern $P_{i,j} = \perp$ designates any empty list where any available sequence on input j is acceptable. The pattern $P_{i,j} = [*]$ is acceptable for any sequence *containing at least one token*. The length of a pattern $P_{i,j}$ is denoted $|P_{i,j}|$.

An actor $a \in V$ can fire when at least one of its firing rules $\mathbf{R}_i \in R_a$ is satisfied. As a result, the DPN model introduces non-blocking read to the semantic of the FIFO channel. So that, an action can be executed if and only if the input data available allow its entire execution. When several firing rules are satisfied at the same time, a single one is chosen based on predefined priorities.

All along this paper, we consider only video decoders even if our approach can be applied to any data-intensive applications. The application complexity has to justify the use of dynamic dataflow modeling over more restricted dataflow modeling that could allow more efficient implementations.

2.3 Reconfigurable Video Coding

Few years ago, MPEG has introduced an innovative framework, called Reconfigurable Video Coding (RVC) [21], that can be considered as the first large-scale experimentation on dynamic dataflow programming. RVC has been initially introduced to overcome the lack of interoperability between the various video codecs deployed in the market. The framework allows the development of video coding tools, among other applications, in a modular and reusable fashion thanks to a dataflow programming language, and the support of a complete development environment known as Orcc [33].

```
1  actor Abs() int I => uint O:
2   pos: action I:[u] => O: [u] end
3   neg: action I:[u] => O: [-u]
4    guard u < 0 end
5   end
6
7   priority
8    neg > pos;
9   end
10 end
```

Listing 1: Description of the absolute value actor in RVC-CAL

The RVC framework includes a subset of CAL programming language [11], known as RVC-CAL, to describe the behavior of the components of the application, i.e. the actors, following the semantic of the dynamic dataflow models. This language is a mixture between imperative and functional programming languages that introduces useful abstractions for dataflow programming. Comparing to the original CAL language, RVC-CAL provides a precise type-system as well as some practical features. The execution of an actor is composed of a sequence of ordered steps, applied repeatedly:

1. First, the actor consumes, or not, a given amount of data from its input ports.
2. Then, it may modify its internal state.
3. Finally, it produces, or not, a given amount of data to its output ports.

As a consequence, describing an actor execution, such the computation of the absolute value presented in Listing 1, involves the description of its interface such as the input ports (I) and the output ports (O), its internal state that is modeled by a set of state variables, as well as the procedural description of the computational steps and the internal scheduling that ordered these steps (guards, priorities, etc).

2.4 RVC-based video decoders

The RVC working group has developed, in parallel with the standardization process, some descriptions of MPEG video decoders using the RVC framework, such as the HEVC description which is presented in Figure 2. In fact, since the standardization of H.261, all existing ITU/MPEG video codecs have globally kept the same

structure [23]. The difference between the standards comes mainly from the evolutions of the algorithmic part that offer an increasing compression rate. As a result, the application graphs of all RVC-based video codecs are quite similar to the structure of our HEVC decoder [21]. The description is decomposed in 4 distinct parts:

1. The first part, called *parser*, extracts values needed by the next processing steps from the coded bitstream. Entropy decoding techniques are used to extract syntax elements whose values are then transmitted to actors that are concerned.
2. A second part, known as *residual*, decodes the error resulting of the image prediction using inverse transforms, such as the well-know IDCT. The transforms allow spatial redundancy reduction within the encoded residual image.
3. A next part, called *prediction*, performs the intra and inter prediction. Intra prediction is done with collocated blocks in the same picture whereas inter prediction is performed as a motion compensation with other pictures. The inter prediction also implies the use of a buffer containing decoding pictures to be able to perform the temporal prediction.
4. And, a last part, called *filters*, reduces the impact of the prediction on the image rendering. For example, the DeBlocking Filter (DBF) is used to smooth the sharp edges between the macroblocks to improve the quality of the decoded image.

RVC-based video decoders are described with an average granularity (at block level), contrary to the traditional coarse-grain dataflow (at frame level). On the one hand, this fine-grain streaming approach induces a high potential in pipeline parallelism and the use of small communication channels, usually sized between 512 and 8192. On the other hand, a finest granularity increases the cost of synchronization between the actors.

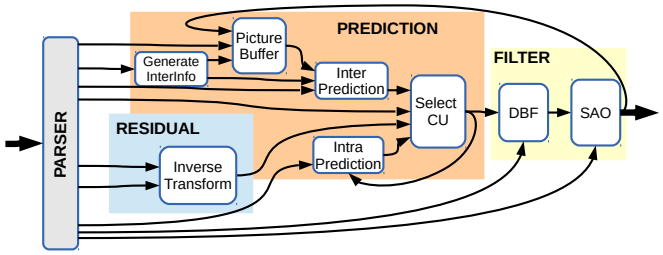


Fig. 2: RVC description of an HEVC decoder

To increase the parallelism exposed within the decoder, the parser can separate the processing of each image components, luma and chroma, in three parallel paths (Y, U and V). The image components are then merged back at the end of the processing. Table 1 summarizes the properties of the experimented descriptions of video decoders: Respectively, the name of the standard, the profile of the decoder, the parallelization of the decoding for each component, the number of actors and FIFO channels.

Codec	Profile	Version	Actors	FIFOs
MPEG-4 Visual	SP	Serial	15	38
		Parallel	39	104
HEVC	Main	Serial	12	83
		Parallel	25	185

Table 1: Statistics about the RVC-CAL description of several MPEG video decoders

3 Architecture model

The development of a design flow targeting embedded multi-core platforms requires the definition of an architecture model that matches the behavior of the targeted platform, while keeping a high-level of abstraction and enough configuration options to allow design-space exploration. Alternatively, architecture models can be presented as customizable multi-core processor templates that setup the main architectural aspects.

Considering the complexity of multi-core architectures, together with the efficiency and the reliability required by embedded systems, we propose to specialize our architecture model for the execution of dynamic dataflow programs in order to take advantage of the knowledge inherent to our application domain.

3.1 Processor Architecture

The processor cores underlying our abstract platform is based on a VLIW-style architecture known as Transport-Trigger Architecture (TTA) [9]. TTA processors resemble VLIW processors in the sense that they fetch and execute multiple operations statically each cycle. Thus, TTA processors are able to take advantage of the low-level parallelism while dataflow models expose explicitly high-level parallelism. A major difference with VLIW processor, however, is that TTA processors have only one instruction: *move*, which simply transfers data from a processor internal place to another one. As a result, the data transports between the register files and the

function units are exposed similarly to the data stream between the components of dataflow models.

Moreover, TTA processors are ideal for targeting embedded systems. Corporaal states that direct programming of the data transports reduces the register file traffic when compared to VLIW [9], but however makes the compiler design quite challenging, as it is the compiler that schedules the data transports and makes sure conflicts are avoided. Since the compiler makes these decisions at design time, the run-time system is simplified and hence there are savings on the processor gate count and energy consumption.

As an example, Figure 3 presents a simple TTA-based processor composed of three buses, one ALU, one multiplier, one register file (RF), one load/store unit (LSU) to manage RAM accesses, and one control unit connected to the ROM containing the instructions. Like most modern processors, TTA processors are based on the *Harvard* architecture that physically separates storage and pathway for instructions and data.

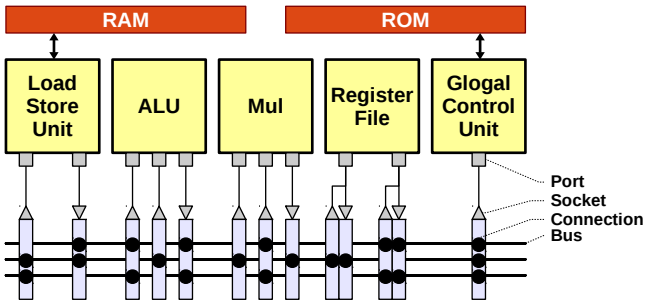


Fig. 3: A simple processor based on Transport-Trigger Architecture

Moreover, the TTA-based Co-design Environment (TCE) makes TTA processors extremely configurable [13]. The TCE is a toolset for designing custom TTA processors which includes a flexible compiler. The designer can make the processor tiny and energy-efficient or, if needed, increase the instruction-level parallelism of the processor.

3.2 Predefined Configurations of Processors

Table 2 presents 4 predefined configurations of TTA-based processors used during our experiments (respectively *Standard*, *Custom*, *Fast* and *Huge*). The configurations characterize internal aspects of the processors such as the number of fonctional units (FUs), ALUs, multipliers and LSU, the number of integer and boolean RFs as well as the number of registers they contain, and the number of buses that interconnect all together FUs and RFs. The connectivity of the interconnection network is also characterized as *Full* or *Custom*. While a *Full* connectivity does not limit the data movement between FUs and RFs, a *Custom* connectivity avoids the decrease of the clock frequency when the complexity of the interconnection network increases.

Processor	*Standard*	*Custom*	*Fast*	*Huge*
ALUs	1	2	3	12
Multipliers	1	1	1	8
LSUs	1+	1+	1+	2+
Int RFs (32bits)	2x12	3x12	3x14	8x32
Bool RFs (1bit)	1x2	1x2	1x6	1x6
Buses	3	6	18	32
Connectivity	Full	Full	Custom	Full

Table 2: Comparison of 4 predefined processor configurations

The first processor configuration, called *Standard*, is almost equivalent to a RISC processor: inside the TTA processor the interconnection network is composed of 3 buses that can provide two operands to the FU at each clock cycle and move the result when it is available. The 3 last configurations, *Custom*, *Fast* and *Huge*, define larger processors composed of several FUs and buses able to take advantage of the instruction-level parallelism of the application (like a VLIW processor). Concerning the *Huge* configuration, its characteristics are deliberately over-sized to acquire the maximal performance, so this configuration is only used in simulation purposes. The *Fast* configuration, introduced in [13], provides clustered TTA-based processors that can reach high-frequency with large potential of parallel computing. We assume that a chip composed of *Fast* TTA processors can reach 1GHz using 40nm CMOS technology such as demonstrated in previous work [18].

3.3 Dataflow-specific Memory Architecture

Now, we introduce an hybrid memory architecture specially designed for dataflow programs. To limit the traditional memory bottleneck, our architecture model contains both shared and private memories. As shown in Figure 4, the processors $(P_1, ..., P_k)$ have their own private memories $(M_1, ..., M_k)$ used for executing their actors, but the processors are also connected, through an interconnection network, to a set of shared memories $(S_1, ..., S_n)$ devoted to inter-processors communications.

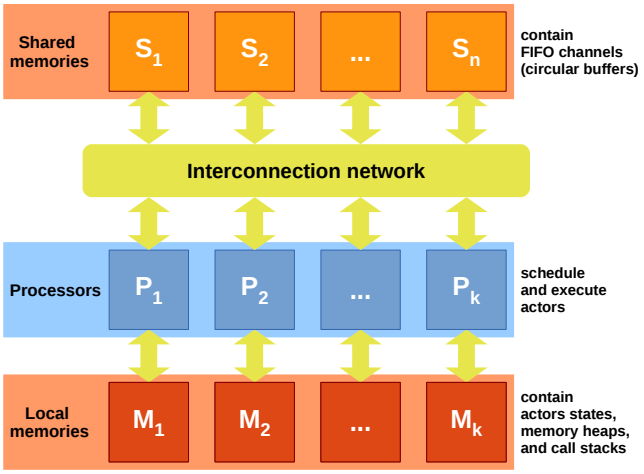


Fig. 4: An hybrid memory architecture dedicated to DPN-based programs

Modeling multi-core platforms dedicated to the execution of DPN-based programs [20] allows us to make the following assumptions: Actors can only communicate through communication channels. Thus, shared memories do not need to store data apart from the content of FIFO-based communication channels, implemented as circular buffers that are detailed later in Section 4. However, the FIFO are mapped to local memory when the two actors are mapped to the same processor. Moreover, the DPN model allows stateful actors. Thus, local memories may have to store the current states of the actors that are assigned to the processor to which they are related. Additionally, local memories have to store the heap and the call stack used during the execution of the actions just as traditional programs.

In comparison with the global shared memory architecture used in most general-purpose processors, this hybrid memory architecture aims to take advantage of the explicit communication of dataflow model to separate the local information from the communications. As a result, data congestion is globally reduced so we assume no conflict at all. Additionally, this architecture reduces the power consumption of the chip since several smaller memory components usually consumes less power than a monolithic centralized memory component [22].

Moreover, storing communication channels in shared memory increases the flexibility of the platform. Knowing that a single memory component can contain multiple channels, the compiler has to assign not only actors to processors but also FIFO channels to memory components. Actually, FIFO channels can be freely mapped to memory components since they are not dependent from each other. But, some architectural constraints may have to be considered, such as the topology of the interconnection network or the size of the memory components.

4 Software synthesis of dynamic dataflow programs

The main challenge that dynamic dataflow programs have to face is the demonstration of efficient implementations that can achieve performance constraints imposed by modern applications. For instance, video decoders have to provide real-time frame-rates for high-definition video sequences.

For that reason, this section presents a set of advanced software synthesis techniques based on preliminary work [34] that enhance the performance of the implementation of dynamic dataflow programs using their specific properties and the flexibility of software systems.

4.1 Specific FIFO channels

In theory, the DPN model defines FIFO channels with unbounded capacity [20]. In practice, the FIFO channels are bounded to limit memory usage and avoid the overhead of dynamic memory allocation. Actually, bounded FIFO channels have been studied extensively, but the DPN model has specificities that make their implementation quite challenging. An action is fired if and only if its *firing rule* is valid. Thus, the implementation of FIFO channels for DPN-based programs requires the ability to check their state, i.e. the number of tokens available, and to *peek* tokens from input channels, i.e. checking values of incoming tokens without consuming them, to evaluate action fireability and thus break conventional FIFO principle.

Now, our dataflow applications also support broadcasting communication following the 1-*producer* / *N*-*consumers* scheme. Thus, actors can produce data that are transmitted simultaneously to multiple target actors through a single port. In fact, the implementation of the broadcasting is another critical point of communication in dynamic dataflow programs, especially for our video decoding applications that have an extensive use of broadcasting. As a result, the implementation of our communication channels has to be able to efficiently broadcast the data over several actors.

4.2 Branch-Free Communications

In software, FIFO channels are traditionally implemented by a circular buffer allocated in a shared memory. *Read*

and *write* are then achieved by accessing the buffer according to read and write indexes that are updated afterwards. Moreover, the comparison of the indexes is sufficient to know the state of the FIFO channel. Finally, a *peek* is a *read* without the update of the read index, but any token can be peeked thanks to the full accessibility of the shared memory. Using circular buffer to implement FIFO channels avoids side shuffles of data after each reading, but implies an advanced management of memory indexes that can ultimately lead to poor performance. For instance, the update of the indexes may require checking if the end of the buffer is reached to go back to the beginning.

```
transp: action
 IN:[ src ] repeat 16 // Input pattern
  ==>
 OUT:[ dst ] repeat 16 // Output pattern
var
  int(size=16) dst[16] =
    [ src[ 4 * column + row ] :
      for int row in 0 .. 3,
       for int column in 0 .. 3
    ]
end
```

Listing 2: Transposition of a 4x4 block in CAL

Avoiding checks on the position of the indexes is however possible using absolute indexes with the cost of additional modulo operations. Thus, performing *read* and *write* increases the indexes infinitely until the overflow of the variables. Since computing the modulo is costly on most processor architectures, it is translated to a simple right shift by forcing the size of the buffer to a power of two. Paradoxically, such a constraint on the size of the communication channels does not have a large impact on the memory usage, especially compared to the large needs of video decoders. Indeed, the initial sizes of our FIFO channels being reasonable, the round-up to the next power of two is relatively small.

Broadcasting tokens can be implemented in two ways according to the locations of the targets:

1. Asking the source actor to broadcast itself the tokens into multiple communication channels: While the implementation is natural, the data are copied for each target.
2. Using circular buffers with multiple read indexes, the smallest one being the global index: While this implementation reduces the data movements to maximum, the managing of the FIFO channels is complicated and all the FIFO channels need to be mapped on the same address space.

4.3 Copy-Free Communications

One of the high-level features of CAL is its ability to describe *multi-rate* actions [11], i.e. actions reading and writing pools of data at each firing, such as the very simple example presented in Listing 2, a transposition of 4x4 pixel block, that reads and writes 16 tokens by firing. In fact, multi-rate actions are common for video coding since the pictures are usually processed block after block. Following this semantic, the body of a multi-rate action, such as the one described in Listing 2, is translated into a function composed of 3 steps as follows [24, 29]:

1. **Reading:** Incoming tokens are read *in order* from the input FIFO channels and stored into the local variables referenced by the input *pattern*. E.g., in Listing 2, 16 tokens are read from the input port `IN` and stored in the local array `src`.
2. **Processing:** The action is processed, as defined in its CAL description, using the local variables referenced into the input and output *patterns* as interfaces. As a consequence, the processing of data is not necessarily described *in order*.
3. **Writing:** Outgoing tokens are written *in order* from local variables referenced by the output *pattern* into the output FIFO channels. E.g., in Listing 2, 16 tokens are written successively from the local array `dst` to the output port `OUT`.

While this implementation stays respectful of the FIFO principle, with the exception of the *peeking*, it also involves two additional copies between the circular buffers and the local variables (knowing that only one copy is mandatory).

```
void transp() {
 int indSrc, indDst;
 for(int row = 0; row<=3; row++) {
  for(int col = 0; col<=3; col++) {
   indSrc = (IN->rdInd + (4*col+row))   %
       IN->SIZE;
   indDst = (OUT->wrInd + (row*4+col))  %
       OUT->SIZE;
   OUT->buff[indDst] = IN->buff[indSrc];
  }
 }
 IN->rdInd += 16;
 OUT->wrInd += 16;
}
```

Listing 3: Copy-free and branch-free action

Since our FIFO channels are implemented in shared memory without access restriction, we can remove all the additional copies to local buffers by accessing directly to the content of the FIFO channels within the processing of the action. So, accesses to input and out-

put variables, such as `src` and `dst`, are replaced by direct accesses to FIFO channels, such as `IN` and `OUT` respectively. Unfortunately, *race conditions*, i.e. synchronization issues, can occur when the action processing does not ensure that the FIFO accesses are performed in order (such as the accesses to `src`). But, the DPN model defines an action firing as a quantum of execution [20], in other words an action firing is an atomic step that cannot be interrupted. Thus, the FIFO indexes can be updated just once at the end of the action without changing the semantic of the application, such as presented in Listing 3. Then, the implementation stays respectful of the FIFO principle of the DPN model. Indeed, other processors cannot access the FIFO rooms involved by this processing since the FIFO indexes are not updated until the action is entirely processed.

To summarize, the three first steps of action firing (Reading, processing, and writing) can be merged together, reducing the memory footprint and the number of instructions to implement the action, as long as the FIFO indexes are updated after the action processing, and thus let the other actors using newly produced data and newly released rooms.

4.4 Aligned Communications

Our branch-free implementation prevents potential optimizations due to absolute indexes. In fact, the compiler cannot know if the access are aligned in the memory or if the end of the circular buffer is reached during the execution of the current action. Thus, we generate two versions of all actions, standard (Listing 3) and aligned (Listing 4), that are executed according to the current position in circular buffers. Only two versions are generated to limit the scheduling overhead, even for more complex actions that may access to multiple inputs and outputs. Moreover, the accesses can be considered always aligned when the production/consumption rates of the associated actions match with the size of the FIFOs.

The aligned version of the action is called whenever the tokens are linearly accessible in all the buffer. So, the relative indexes can be considered as invariant in order to be computed only once at the beginning of the action (similar to loop-invariant code motion). Additionally, the aligned accesses to the circular buffer are vectorizable since the width of the FIFO channels within our applications are often inferior to the bus width (8 or 16 bits are common values in video processing). As a result this optimization is very powerful for processors that exploits instruction-level parallelism and word-level parallelism.

```
1  void transp_aligned() {
2   int IN_rdInd = IN->rdInd % IN->SIZE;
3   int OUT_wrInd = OUT->wrInd % OUT->SIZE;
4   int ind_Src, ind_Dst;
5   for(int row = 0; row<=3; row++) {
6    for(int col = 0; col<=3; col++) {
7     indSrc = IN_rdInd + (4*col+row);
8     indDst = OUT_wrInd + (row*4+col);
9     OUT->buff[indDst] = IN->buff[indSrc];
10   }
11  }
12  IN->rdInd += 16;
13  OUT->wrInd += 16;
14 }
```

Listing 4: Aligned action

4.5 Multi-level Dynamic Scheduling

As defined by Lee and Parks [20], the execution of a DPN-based actor is modeled by the repeated evaluation of the firing rules that are, in case of a success, followed by the firing of the associated action. This process is usually defined as the *action scheduling*. The action scheduler can be implemented by a simple function that evaluates the firing rules *in order* [29] such as presented in Listing 5. In theory, the scheduler evaluates only two conditions to determine the fireability of an action: the amount of tokens required in the input channel (`hasTokens`), and the potential condition on the values of tokens and/or state variables (`isSchedulable`). In practice, the scheduler has also to ensure that enough rooms are available in the output channels to allow the firing of the action without blocking (`hasRooms`).

Additionally, the scheduler checks if a sufficient number of tokens are aligned in all the FIFO channels to be able to execute the optimized version of the action (`areAligned`). In some specific cases, we can directly insure that the FIFO accesses will be always aligned. As an example, the alignment is guaranteed when the consumption/production rates are constant and divisor of the size of the FIFO channel.

Apart from this internal scheduling, the execution of a DPN program in a concurrent environment requires actor scheduling to order and time the actor execution in case there is more actors than processors. In previous works [32, 31], we have introduced run-time actor mapping/scheduling strategies dedicated to DPN-based actors. Our scheduling strategies execute the current actor until it cannot fire anymore to exploit spatial and temporal locality. Then, the scheduler switches to the next actor which is chosen according to the strategy.

To conclude, the execution of DPN-based programs involves a complex scheduling that has to be performed

at runtime. While they are two distinct levels of scheduling, *actor scheduling* and *action scheduling*, they are intimately related since the success of the action scheduling within an actor is directly dependent on the production/consumption performed by its predecessors/successors. These schedulers have to be carefully designed to not reduce dramatically the performance since they are executed at run-time.

```
1  void Transpose4x4_0_scheduler() {
2   while (1) {
3    if (hasTokens(fifo_Src, 16) &&
         isSchedulable_transp()) {
4     if (hasRooms(fifo_Dst, 16)) {
5      goto finished;
6     }
7     // Fire the action
8     if (areAligned(fifo_Src, 16) &&
          areAligned(fifo_Dst, 16))
9      transp_aligned();
10    } else {
11     transp();
12    }
13   } else { // Check the next action...
14    goto finished;
15   }
16  }
17 finished:
18   return; // Return to actor scheduler
19 }
```

Listing 5: Action scheduler

5 Results

This section studies the implementation of dynamic dataflow programs on TTA-based multi-core platforms. In general, communication and synchronization are the major sources of inefficiencies on every multi-core system. Thus, we deeply analyze the internal behavior of the applications (communication, decomposition, etc) before presenting the global performance.

5.1 Experimental setup

The software implementations are generated by use of the TTA back-end of Orcc [30], then the generated code is compiled and simulated thanks to the TTA-based Co-design Environment (TCE) [13]. The evaluation is made thanks to the instruction-set simulator including in the TCE.

The experiments have been conducted for some of the RVC descriptions of video decoders that have been introduced in Section 2.4, and using different video sequences. During all our experiments, all the FIFO channels in our applications are bounded to 8192 elements in order not to impact on the results. In fact, this specific size of FIFOs allows the buffering of two of the biggest pixel blocks defined in the HEVC standard, i.e. Coding Tree Blocks containing 64x64 samples.

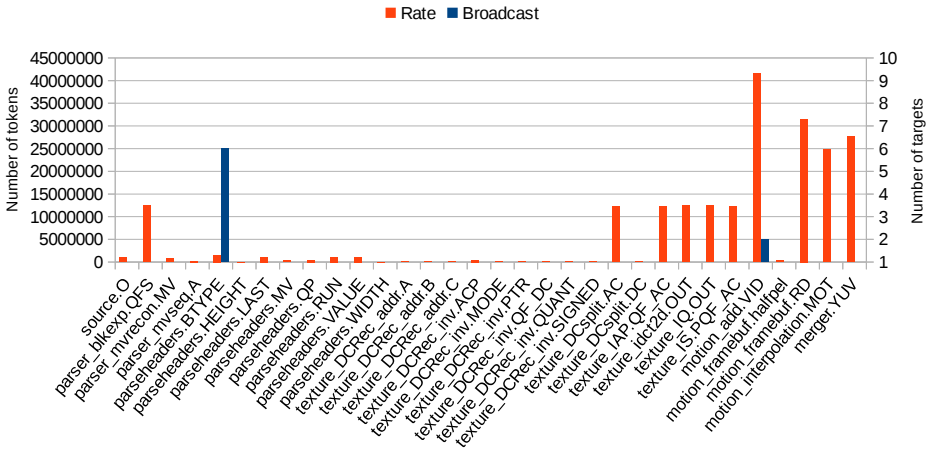
5.2 Analysis of Internal Communications

A major interest of dataflow programs is the explicit communication between the components of the application that makes them easier to analyze. In DPN-based video decoders, communication rates are usually irregular and very sensitive to multiple factors (size of the FIFO channels, actor scheduling, etc). But, communication rates become globally stable when the observed time-slice is sufficient.

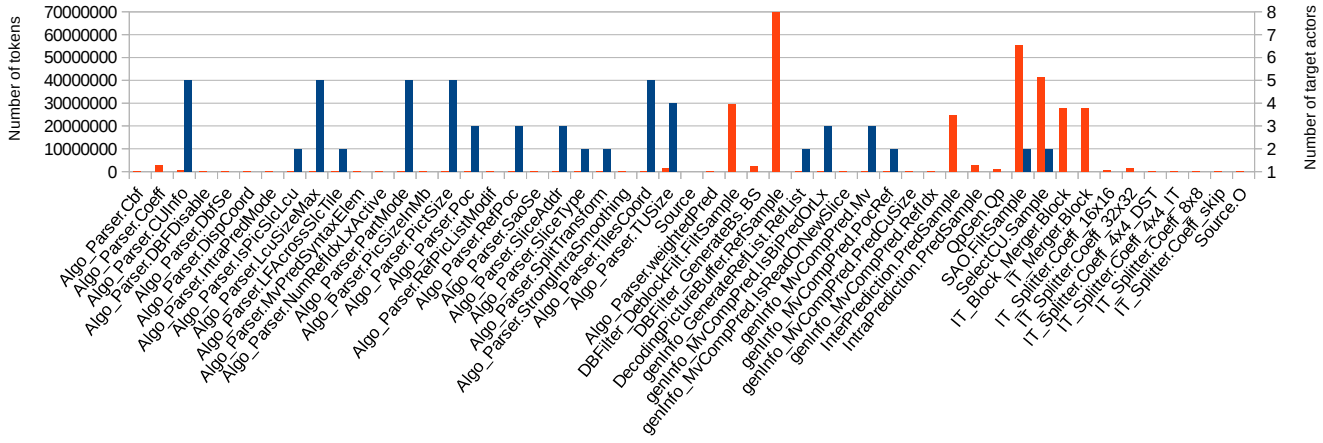
Figure 5 presents the communication rate observed at each output port of actors within the MPEG-4 Visual and HEVC decoders during the decoding of few frames of the tested video sequences. Figure 5 additionally presents the degree of broadcasting of the actors ports, i.e. the number of actors to which the ports are connected, in order to highlight the duplication of data.

We can clearly identify two categories of communications from the results presented in Figure 5:

- The **video stream** is characterized by a large amount of data that usually goes through the decoder by a single path (for instance `parser_blkexp.QFS` in Figure 5a). Besides, broadcasting the video stream involves a large amount of data duplication but is only performed one or two times (For instance `motion_add.Vid` in Figure 5a), when the decoded frames are transmitted to both the display and the image buffer used by the inter prediction. This stream being clearly the largest of the application, this specific broadcast can be the cause of a data congestion.
- The **control communications** are characterized by a small amount of data disseminated through multiple channels within the video decoder. A typical example is the transmission of the type of the current block, `parseheader.BTYPE` in Figure 5a. A major part of these communications is produced by the *parser* which extracts the syntax elements from the input stream to parametrize the actors. As opposed to the video stream, broadcasting the control information implies a smaller amount of data but more consumers. For example, control tokens generated by the *parser* may be transmitted to most of the next actors, like `Algo_Parser.CUInfo` in Figure 5b, so even a small amount of data can introduce a lot of checks to control the state of the communication channels.



(a) MPEG-4 Visual



(b) HEVC

Fig. 5: Communication analysis (rates and broadcasting) within RVC-based video decoders

To sum up, the video stream is processed block after block through the actors which behave according to control data. Moreover, the broadcasting may be an additional source of bottlenecks, causing either data congestions or management overheads.

5.3 Analysis of the Application Decomposition

Now, let us take a look at the application decomposition which is fundamental for targeting multi-core platforms. Indeed, we need to balance the computation load on the available processors to fully benefit from the parallelism.

Workload distribution We start by analyzing the distribution of the computational workload within the video decoders, i.e the computational workload of the actors. The results for two video decoders, MPEG-4 Visual and HEVC, are presented in Figure 6. The workloads are evaluated for each actor independently in a standalone simulation. In other words, each actor is simulated on its own processor with all incoming data available, in order to hide the impact of the stream dependences within the network.

The results clearly show that our description of MPEG-4 Visual is more equitably balanced than our description of HEVC. This difference can be partially explained by the difference between the applications granularity, for instance the *inverse transforms*, designed with 1 actor (the *IDCT2D*) in MPEG-4 Visual and with 12 actors (the *ITs*) in HEVC.

Moreover, it should be noted that the computational workload could be balanced more equitably by increas-

ing the coarse-grain parallelism in the decoder. In video decoding, increasing the parallelism is usually achieved by separating the decoding of the image components or by splitting the image. On the one hand, the separation of the processing of the components is bounded by the luma processing which is four times the complexity of each chroma processing. On the other hand, the decomposition of the image itself is restrained by the spatial and temporal dependences resulting of the prediction. Actually, parallel processing is one of the main achievement of the emerging HEVC standard [27] that introduces several advanced decomposition (wavefront, tiles, etc).

Internal parallelism Thanks to the flexibility of TTA processors in our design flow, we can also study the potential parallelism within the actors. In fact, the predefined processor configurations, presented in Section 3.2, have all their own parallel processing capability, which let us study the ILP potential within actors. Therefore, Figure 7 presents the execution speedup of actors of the two video decoders on *Custom*, *Fast* and *Huge* processors according to their execution time on a *Standard* processor. As said previously, the *Standard* processor is equivalent to a RISC processor that can only perform one operation at a time because of its 3 buses. The actors are again executed in a standalone fashion to hide stream dependence.

The results clearly show two types of actors. On the one hand, actors that benefit well from the parallel capabilities of TTA-based processors by presenting impressive speedups that reach factors up to 3, such as the one processing the inverse transform. We define them as the *compute-intensive* actors. On the other hand, actors that do not take advantage from the parallel capabilities of TTA-based processors by presenting speedups that hardly reach factors of 1.5, such as the ones involved in entropy decoding. We define them as *control-intensive* actors. However, some actors of the HEVC decoder that are known to be compute-intensive have not demonstrated large speedups, such as the predictions and the loop filters. This can be explained by the development state of the application.

From all these results, we can identify the traditional *bottleneck* actors of our RVC-based video decoders: The *parser* that is controlled by a complex scheduling (e.g. the parser of our HEVC decoder contains about 200 actions), the *buffer* which is usually strangled by the number of hardly predictable memory accesses, and finally the *predictions* as well as the *loop filters* that all involve complex processing requiring careful implementations. In conclusion, video decoders are now complex applications containing heterogeneous algorithms which make their implementation so challenging.

For that reason, the actor mapping system included in our design flow considers both the communication rates and the computational decomposition for the design decisions, as explained in Section 5.4.

5.4 Analysis of performance

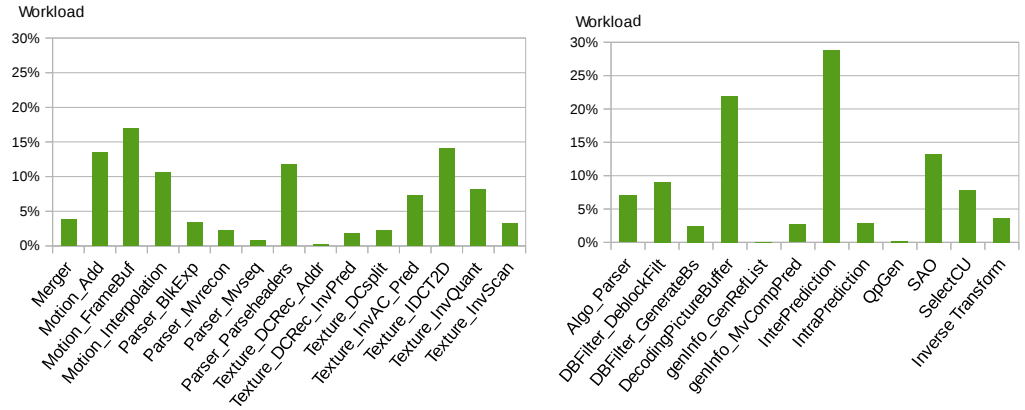
Finally, we analyze the global performance of our RVC-based video decoders. Let us point out that a functional implementation of a video decoder running on an embedded multi-core platform is very difficult to obtain. Indeed, debugging dataflow programs within embedded multi-core platforms is a hard and time-consuming task that requires an expertize from hardware and software aspects. Moreover, the simulation speed is rapidly becoming one of the main limitations in front of the application complexity.

The evaluated platforms are composed of *Fast* TTA processors interconnected by shared memories following the architecture defined in Section 3. We assume that such platforms can be clocked at 1GHz. Indeed, previous work has shown that the processor cores can already reach 1GHz using 40nm technology [18]. Thus, the results are obtained from a simulated execution, but let us point out that successful implementations of the MPEG-4 Visual decoder has already been synthesized on two different FPGA boards clocked at 100MHz: Altera Stratix III and Xilinx Virtex 6.

Maximal performance Table 3 summarizes the maximal decoding frame-rates achieved with our implementation on both the MPEG-4 Visual decoder and the HEVC decoder. In order to get the maximal performance, each actor is mapped to its own processor. Thus, there is no need for an actor scheduling strategy: The global scheduling is achieved by the action scheduler that checks repetitively the validity of the firing rules.

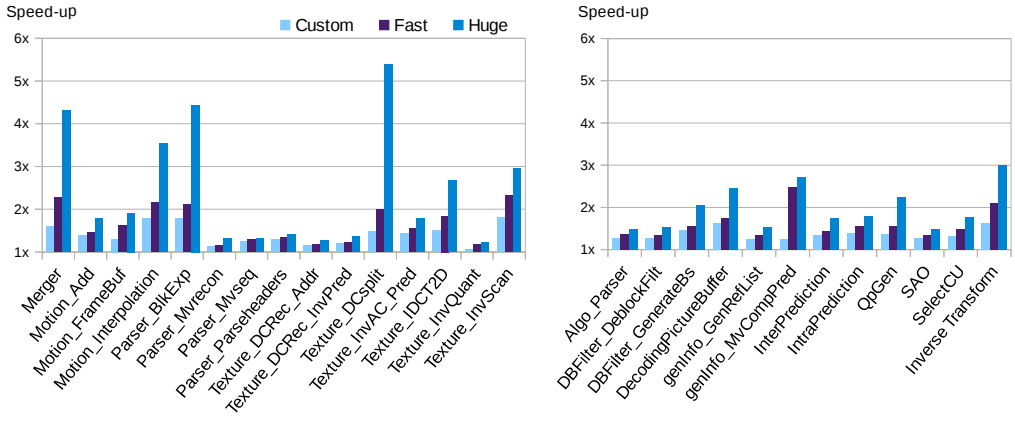
Besides the functional demonstration, the results also show a large difference of performance between the two decoders, i.e. the frame-rate observed on MPEG-4 Visual is about 8 times better on sequences with identical definition. This can be explained by the performance tuning that we have already made on the description of MPEG-4 Visual, along with the algorithmic complexity of the new standard and the development status of our description of HEVC. Considering the current performance, our implementation of HEVC cannot achieve real-time decoding of high definition sequences.

However, these results open promising perspectives about a more optimized implementation, that would



(a) MPEG-4 Visual (b) HEVC

Fig. 6: Repartition of the computational workload within our implementations of RVC-based video decoders



(a) MPEG-4 Visual (b) HEVC

Fig. 7: Exploring the parallelism potential of actors composing video decoders thanks to their execution speedup on TTA-based processors using *Custom*, *Fast* and *Huge* configurations from a sequential execution with *Standard* configuration

Decoder	Sequence	Size	FPS
MPEG-4 Visual	*Foreman*	QCIF	1750
– 39 processors	*OldTownCross*	720P	40
HEVC	*BasketBallPass*	240P	40
– 12 processors	*KristenAndSarah*	720P	5

Table 3: Maximal frame-rates achieved by our embedded implementation using the *Fast* TTA configuration clocked at 1GHz when each actor is mapped to its own processor. These frame-rates have been evaluated during an execution of the entire multi-core platform using the instruction-set simulator.

include highly optimized assembly kernels (like most commonly-used video codecs [15]). Knowing the high parallel processing capabilities of TTA processors, such assembly-level optimization can speed-up the decoding suffisently to achieve real-time decoding. Moreover, processing resources can be shared between the actors to reduce the number of processor without impacting too much the performance, as shown by the following paragraphs.

Influence of the core number Now, let us take a look at the influence of the number of processors available on the platform. In fact, some of the actors have to share the same processor in realistic implementations. Indeed, the number of processors available must be limited so as to reduce the power consumption of the platform.

As opposed to the previous experimentation, the actors are mapped by an automated system [31] which takes into account the irregularity of our applications thanks to a profiling step, as presented in Figure 8. Our

mapping system starts by analyzing the communication rates and the computational loads, as we did respectively in Section 5.2 and Section 5.3. Then, the system tries to balance the computation load of the actors to parallelize the work while reducing the inter-core communications. For this purpose, we use multi-level graph partitioning schemes implemented in Metis tool [17]. In other words, two actors communicating a lot with each other have more chance to be executed on the same processor. Finally, the actors are scheduled locally on each processor core by a simple runtime strategy known as round-robin [32].

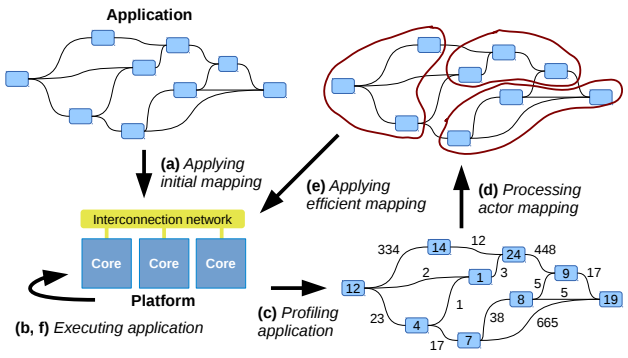


Fig. 8: Actor mapping system based on computation and communication analysis [31]

Figure 9 presents the influence of the number of processors on the frame-rate of the MPEG-4 Visual decoder. In this case, we consider the decoding of a video sequence with a smaller definition, i.e. *foreman* at QCIF resolution, to reduce the simulation time. The decoding is again simulated using the *Fast* configuration for the TTA processors.

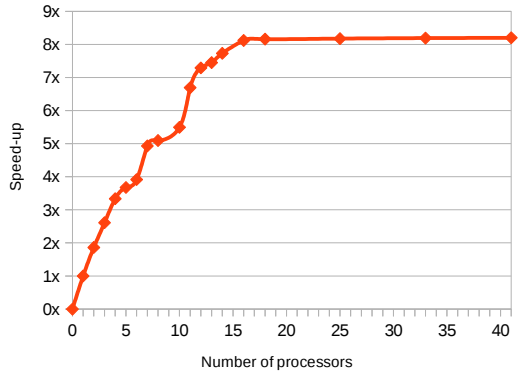


Fig. 9: Influence of the number of processors on the performance of MPEG-4 Visual decoder

First of all, the results clearly show that the acceleration rate is not linear according to the number of cores. In fact, the form of the curve clearly shows the limit of the coarse-grain parallelism (task-level) of the application. Actually, the maximum decoding frame-rate of our MPEG-4 Visual decoder is reached with 16 processors. Increasing further the number of processors does not provide higher decoding frame-rate. These results can be explained by the complexity of the data dependencies in video decoding (spatial and temporal). Higher parallelism can be achieved thanks to parallel decoding technics (framebase, tiles, wavefront, etc).

Thus, the maximum speedup in comparison with the single processor execution is 8.1x, and achieved with 16 processors. Therefore, the maximum speedup achieved with our embedded implementation is much bigger than the maximum speedup achieved with the implementation on general-purpose processors (which seems to be around 3x [31]). This can be mainly explained by the fact that the communications between the cores within our embedded implementation do not induce any overhead compared to more conventional communication and memory schemes implemented in general-purpose processors. To conclude, these results demonstrate the interest of the dedicated memory organization that we have designed specifically for our custom embedded multi-core platforms (see Section 3.3).

6 Related work

Implementing video codecs using dynamic dataflow modeling has already been heavily studied within the RVC community. However, most of the studies do not target multi-core platforms based on distributed memory organization, but platforms such as FPGA/ASIC [4, 1, 25] and general-purpose processors [29, 14]. In previous work [30], we have already implemented an MPEG-4 Visual decoder on a platform composed of TTA processors interconnected by hardware FIFO channels. This approach targets application-specific platforms which makes it much less flexible than our new approach. Outside of the architecture side, our software synthesis which is also applicable on general-purpose processors has significantly improved the performance: We observed an improvement of more than 100% of the decoding frame-rates over previous implementations (at equal frequency) [34].

Other studies from the literature try to improve the predictability of dynamic dataflow programs so as to allow compile-time optimizations. Some of them determine the possible executions to prune all unreachable execution paths in order to remove all unnecessary tests

[7, 12]. However, they are limited by their need of input data to perform their analysis, which makes them unsafe in general case. Some other approaches try to reduce the number of tests performed during the scheduling by detecting restricted dataflow models [28], or by using *actor machines* that also considers the evaluation results of previous firing rules [8]. However, these techniques have not yet demonstrated performance improvements of tested applications.

Regarding the HEVC standard, to our knowledge all existing software decoders are based on multi-threaded implementations, such as the reference software (HM) [6] and OpenHEVC [15]. Multi-threaded implementations assume that the architecture of the executing platform is based on a global shared memory organization. On the one hand, these implementations have been demonstrated very efficient mainly due to the minimization of data movements during the processing. On the other hand, their parallelization is limited since embedded platforms based on shared memory cannot scale beyond a certain number of processor cores because of power consumption.

To sum up, our work tries to bridge the gap between the efficiency of low-level implementations and the flexibility/reliability of high-level implementations in order to facilitate the design of complex applications, such as video codecs, on parallel embedded systems.

7 Conclusion

This paper presents a methodology based on dataflow modeling to implement video codecs on embedded multi-core platforms. We have introduced an architecture model to design low-power multi-core platforms using a distributed memory organization that directly benefit from the dataflow modeling. We have also presented advanced software synthesis techniques to enhance the implementation of dynamic dataflow programs on embedded multi-core platforms using branch-free, copy-free and aligned implementations to tackle communication and computation issues. Our methodology has been validated both on MPEG-4 Visual and HEVC decoders. The results show an improvement of more than 100% of the framerate over previously proposed dataflow implementations, and achieve real-time performance on HD video sequences using the MPEG-4 Visual decoder while keeping a high-level of abstraction.

References

1. Mariem Abid, Khaled Jerbi, Mickaël Raulet, Olivier Déforges, and Mohamed Abid. System Level Synthesis Of Dataflow Programs: HEVC Decoder Case Study. In *Electronic System Level Synthesis Conference (ESLsyn), 2013*, 2013.
2. Mohamed A. Bamakhrama, Jiali Teddy Zhai, Hristo Nikolov, and Teodor Stefanov. A methodology for automated design of hard-real-time embedded streaming systems. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 941–946. Ieee, March 2012.
3. Vagelis Bebelis, Pascal Fradet, Alain Girault, and Bruno Lavigueur. BPDF: A Statically Analyzable DataFlow Model with Integer and Boolean Parameters. In *Embedded Software (EMSOFT), 2013 Proceedings of the International Conference on*, 2013.
4. Endri Bezati, Simone Casale Brunet, Marco Mattavelli, and Jörn W. Janneck. Synthesis and Optimization of High-Level Stream Programs. In *Electronic System Level Synthesis Conference (ESLsyn), 2013*, 2013.
5. Bishnupriya Bhattacharya and Shuvra S. Bhattacharyya. Parameterized Dataflow Modeling for DSP Systems. *IEEE Transactions on Signal Processing*, 49(10):2408–2421, 2001.
6. Frank Bossen, Benjamin Bross, Karsten Sühring, and David Flynn. HEVC Complexity and Implementation Analysis. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1685–1696, 2013.
7. Jani Boutellier, Mickaël Raulet, and Olli Silvén. Automatic Hierarchical Discovery of Quasi-Static Schedules of RVC- CAL Dataflow Programs. *Journal of Signal Processing Systems*, 71(1):35–40, 2013.
8. Gustav Cedersjö and Jörn W. Janneck. Toward Efficient Execution of Dataflow Actors. In *Signals, Systems and Computers (ASILOMAR), 2012 Conference Record of the Forty Sixth Asilomar Conference on*, pages 1465–1469, 2012.
9. Henk Corporaal. *Microprocessor Architectures: from VLIW to TTA*. John Wiley & Sons, Chichester, UK, 1997.
10. Karol Desnos, Maxime Pelcat, Shuvra S. Bhattacharyya, and Slaheddine Aridhi. PiMM: Parameterized and Interfaced Dataflow Meta-Model for MPSoCs Runtime Reconfiguration. In *Embedded Computer Systems (SAMOS), 2013 International Conference on*, 2013.
11. Johan Eker and Jörn W. Janneck. CAL language report: Specification of the CAL actor language. Technical report, University of California, Berkeley, Berkeley, 2003.
12. Johan Ersfolk, Ghislain Roquier, Johan Lilius, and Marco Mattavelli. Scheduling of dynamic dataflow

programs based on state space analysis. In *IEEE International Conference on Acoustics, Speech, and Signal Processing, 2012. ICASSP-12.*, pages 1661–1664, 2012.

13. Otto Esko, Pekka Jääskeläinen, Pablo Huerta, Carlos S. de La Lama, Jarmo Takala, and Jose Ignacio Martinez. Customized Exposed Datapath Soft-Core Design Flow with Compiler Support. In *Proceedings of the 2010 International Conference on Field Programmable Logic and Applications*, pages 217–222, 2010.
14. Jérôme Gorin, Matthieu Wipliez, Françoise Prêteux, and Mickaël Raulet. LLVM-based and scalable MPEG-RVC decoder. *Journal of Real Time Image Processing*, 6(1):59—-70, 2011.
15. Wassim Hamidouche, Mickaël Raulet, and Olivier Déforges. Parallel SHVC decoder: Implementation and analysis. In *Multimedia and Expo, 2013 IEEE International Conference on*, 2013.
16. G Kahn. The semantics of a simple language for parallel programming. *Information processing*, 74:471–475, 1974.
17. George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, January 1998.
18. Heikki Kultala, Otto Esko, Pekka Jääskeläinen, Vladimir Guzma, Jarmo Takala, Jiao Xianjun, Tommi Zetterman, and Heikki Berg. Turbo decoding on tailored OpenCL processor. In *2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 1095–1100. Ieee, July 2013.
19. Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.
20. Edward A. Lee and Thomas Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.
21. Marco Mattavelli, Mickaël Raulet, and Jörn W. Janneck. MPEG reconfigurable video coding. In Shuvra S. Bhattacharyya, Ed F. Deprettere, Rainer Leupers, and Jarmo Takala, editors, *Handbook of Signal Processing Systems*, pages 281–314. Springer, New York, NY, USA, 2013.
22. Jörg Mische, Stefan Metzlaff, and Theo Ungerer. Distributed Memory on ChipBringing Together Low Power and Real-Time. Technical report, University of Augsburg, 2014.
23. Iain E G Richardson. *H.264 and MPEG-4 Video Compression: Video Coding for Next-generation Multimedia*. John Wiley & Sons, Inc., New York, NY, USA, 2003.
24. Ghislain Roquier, Matthieu Wipliez, Mickaël Raulet, Jörn W. Janneck, Ian D. Miller, and David B. Parlour. Automatic software synthesis of dataflow program: An MPEG-4 simple profile decoder case study. In *Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on*, pages 281–286, 2008.
25. Nicolas Siret, Matthieu Wipliez, Jean-François Nezan, and Francesca Palumbo. Generation of Efficient High-Level Hardware Code from Dataflow Programs. In *Proceedings of Design, Automation and Test in Europe (DATE)*, 2012.
26. Sander Stuijk, Twan Basten, Benny Akesson, Marc Geilen, Orlando Moreira, and Jan Reineke. Designing next-generation real-time streaming systems. In *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis - CODES+ISSS '11*, pages 3–4, 2011.
27. Gary J. Sullivan, Jens-Rainer Ohm, Woo-Jin Han, and Thomas Wiegand. Overview of the High Efficiency Video Coding (HEVC) Standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1649–1668, December 2012.
28. Matthieu Wipliez and Mickaël Raulet. Classification of Dataflow Actors with Satisfiability and Abstract Interpretation. *International Journal of Embedded and Real-Time Communication Systems*, 3(March):49–69, 2012.
29. Matthieu Wipliez, Ghislain Roquier, and Jean-François Nezan. Software Code Generation for the RVC-CAL Language. *Journal of Signal Processing Systems*, 63(2):203–213, June 2009.
30. Hervé Yviquel, Jani Boutellier, Mickaël Raulet, and Emmanuel Casseau. Automated design of networks of Transport-Triggered Architecture processors using Dynamic Dataflow Programs. *Signal Processing Image Communication*, 28(10):1295–1302, 2013.
31. Hervé Yviquel, Emmanuel Casseau, Mickaël Raulet, Pekka Jääskeläinen, and Jarmo Takala. Towards run-time actor mapping of dynamic dataflow programs onto multi-core platforms. In *Image and Signal Processing and Analysis (ISPA), 2013 8th International Symposium on*, pages 732–737, 2013.
32. Hervé Yviquel, Emmanuel Casseau, Matthieu Wipliez, and Mickaël Raulet. Efficient multicore scheduling of dataflow process networks. In *Signal Processing Systems (SiPS), 2011 IEEE Workshop on*, pages 198–203, 2011.
33. Hervé Yviquel, Antoine Lorence, Khaled Jerbi, Alexandre Sanchez, Gildas Cocherel, and Mickaël Raulet. Orcc: Multimedia development made easy. In *Proceedings of the 21st ACM international con-*

ference on Multimedia, pages 863–866, 2013.

34. Hervé Yviquel, Alexandre Sanchez, Pekka Jääskeläinen, Jarmo Takala, Mickaël Raulet, and Emmanuel Casseau. Efficient Software Synthesis of Dynamic Dataflow Programs. In *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, 2014.