Integration of Dataflow Components Within a Legacy Video Transcoding Framework

Tewodros Deneke[1,2], Lionel Morel[3], Sébastien Lafond[2] and Johan Lilius[2]
[1]TUCS - Turku Centre for Computer Science, Finland
[2]Åbo Akademi University, Finland
[3]Universite de Lyon, Inria INSA-Lyon, CITI, F-69621, France

***Abstract*—Recently the RVC-CAL dataflow language has enabled video codecs to be specified in a more natural way than imperative languages by allowing implicit expression of parallelism and side effect freeness. The tools developed for RVC-CAL have also enabled the automatic generation of parallel C code, among others, from dataflow specifications.**

This paper introduces a new approach allowing the integration of dataflow components within legacy code. The approach makes use of a generic interface definition that allows seamless interaction between I/O components, which are mostly state operations and are best implemented in imperative languages with data processing components which are mostly stateless dataflow operations and are best implemented in dataflow languages. The advantage of the approach is the ease of development by allowing each language to be used on those parts of the application that it is most appropriate for.

The functionality of the approach is demonstrated by using the generic interface to add a new dataflow based MPEG and HEVC decoder into the legacy video transcoding library FFmpeg.

I. Introduction

Video is becoming an important medium of communication and accounts for a significant portion of the available bandwidth in the world. Today video is being consumed through various devices and networks. These devices and networks have varying capabilities in terms of screen resolution, storage space, processor speed and bandwidth. Video transcoding is needed to enable seamless exchange of videos among various devices on heterogeneous networks like the internet. Video transcoding is the process of transforming one video format with certain characteristics such as bitrate, framerate and resolution to another video with different characteristics that will better fit the target device. One typical scenario of transcoding would be the delivery of video content by video on demand (VOD) systems, like YouTube, to customers connected trough wireless or wireline using their mobile phone, tablet or IPTV. In order to reach such a wide range of consumer segments, VODs transcode and store videos in several formats that are tailored to each segment.

As presented in figure 1, the components of a video transcoder can be roughly categorized in to two: I/O and video processing components. The I/O components are responsible for reading/writing video/audio in different formats to/from a disk or network (e.g. Muxers and Demuxers). The video processing components are mainly responsible for manipulating the video/audio content (e.g, encoder and decoder). While the I/O components are inherently serial and implemented as state operations, the video processing components are parallelizable and compute intensive and are more naturally expressed with a dataflow programming model. For example a video codec, which is one of the video processing components, is a software that enables compression and decompression of digital multimedia content. The main use of a codec is to compress video so that it uses less resources during transmission and storage and later to decompress it for viewing.

Over the years video codecs have become more efficient in terms of their compression ability while becoming more compute intensive. Until recently the increase in computational complexity of codecs has not been a real concern because computer speed has been doubling every other year following Moore's law. However due to physics constraints like power dissipation and transistor scaling, multi-core architectures have become the solution to allow performance to keep increasing. Programming codecs for multi-cores is currently done using imperative languages such as C/C++ using threads to explicitly express parallelism. This is obviously a challenging, time consuming and error prone process as multi-threading is essentially a way to only prune non-determinism after its introduction [1]. Research in dataflow programming shows the advantages of using dataflow languages such as RVC-CAL for the purpose of video codec specification and implementation. Among others, the main advantages of using dataflow as the main programming method include ease of use, flexibility, automatic analyzability, automatic parallelizability, visual presentability and above all side effect freeness [2], [3], [4].

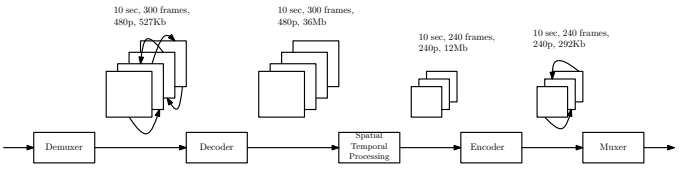

Fig. 1. Basic description of video transcoding. Demuxer and Muxer are I/O components used to read/write compressed video and the rest are video processing components which encode, decode and scale video.

While programming video processing components of a video transcoder (e.g. codecs) entirely in dataflow is an attractive idea, programming the I/O components in dataflow is unnatural on Von-Neumann architecture. I/O operations have been introduced as state operations in the Von-Neumann architecture as a convenient and clean way of reading in data into programs. On the other hand the absence of side effects which is fundamental to dataflow programs does not allow state operations and makes dataflow implementation of I/O operations difficult.

In this paper we propose an approach based on the defini-

tion of a generic interface that enables video/audio processing system designers to write dataflow components in a dataflow language and interface them with existing legacy video/audio applications such as a transcoder. The proposed approach allows the splitting of the development of video processing applications in two parts: one part which uses dataflow language and implements parallel sections and another, which uses imperative languages and implements the I/O sections. It combines the advantages of using dataflow languages with the features of legacy imperative code and helps quicker adoption of dataflow languages.

The rest of the paper is organised as follows. Section II introduces the reader with necessary background knowledge. Section III reports on related works. Section IV actually describes the body of work of our proposition. Section V presents experimental results evaluating the benefits in terms of performance and the seemingless integration of a substantial dataflow application into the FFmpeg transcoding platform. Finally, section VI concludes and gives few perspectives to this work.

II. Background

A. Video Transcoding and FFmpeg

The difference in device resources, network bandwidth and video representation types results in the need for a mechanism enabling video content adoption. This mechanism, called transcoding is currently being used for such purposes as: bitrate reduction in order to meet network bandwidth availability, resolution reduction for display size adoption, temporal transcoding for frame rate reduction and error resilience transcoding for insuring high quality of service (QoS) [5], [6].

As can be noted from figure 2 a generic video transcoder contains five main parts, each having a set of components depicted as blocks. The five main parts include a demuxer, a decoder, spatial and temporal processors, an encoder and a muxer. The demuxer is used to read interleaved streams (e.g. one audio, one video and a subtitle stream) from a network or a file. Usually a set of streams are encapsulated in a container format such as MP4. Packets which are read by the demuxer and contain compressed audio/video frames are then passed to the appropriate decoder to be decompressed. The decompressed audio/video frames are then spatially or temporally processed to adapt the video/audio to a particular framerate and/or resolution. Spatially and temporally processed uncompressed frames are then passed on to the encoder to be compressed via removing temporal, spatial and statistical redundancy that exists inside and among uncompressed frames.

Legacy transcoders such as FFmpeg constitute libraries corresponding to each block. For example FFmpeg contains multiple codec implementations including H264, HVEC, MPEG4. Similarly it contains multiple implementations of muxers and demuxers that can be used to read and write several video and audio file formats such as MP4, MKV, FLV, MP3, WAV, etc.

FFmpeg is an open-source, complete, cross-platform solution to record, convert and stream audio and video[7]. Among others it includes libraries such as libavcodec, libavformat, libavfilter, libavswscale which contain implementations of several video and audio codecs, container formats, filters and spacial processing functions. It is a large project with over 570k lines of code.

The FFmpeg transcoding framework implemented based on FFmpeg libraries provides coarse-grained task (pipeline) and data parallelism. The different components such as the decoder and the encoder can run in a task-based parallel manner. The data parallelism in FFmpeg is currently based on slice-level and/or frame-level threading. In case of slice-level parallelism multiple threads can decode slices (independent parts of a frame) in parallel. In the case of frame-level parallelism multiple threads processes multiple frames. However frames have inter-dependencies and the number of slices in a frame is usually limited to one. Therefore the parallelism implemented in FFmpeg is obviously limited.

B. Dataflow Programming and RVC-CAL

A dataflow program is defined as a directed graph whose vertices are actors (the basic computational units) and edges are unidirectional FIFO channels with unbounded capacity. A stream of data tokens, is processed by actors and passed on to others actors via FIFOs. The advantage of such programming model is its ability to implicitly express concurrency and to enable analyzability. Dataflow graphs we consider here respect the semantics of Dataflow Process Networks (DPNs) [8], which are related to Kahn Process Networks (KPNs) [9]. The main difference between DPN and KPN is that DPN allows actors to check the availability of tokens in the FIFOs. Additionally to the KPN model, DPN introduces the notion of firing. An actor firing, or action, is an indivisible quantum of computation which corresponds to a mapping function of input tokens to output tokens applied repeatedly and sequentially on one or more data streams. This mapping is composed of three steps: input data reading, then computation, and finally output data writing. These functions are guarded by a set of firing rules which specifies when an actor can be fired, i.e. if the number and the values of tokens that are available on the input ports are sufficient.

Dataflow has been used for naturally expressing digital signal processing applications for decades. Currently it has gained particular attention in expressing video processing applications. RVC-CAL is one domain specific dataflow language for video coding based on the DPN model of computation [2], [3], [4], [10]. It was originally developed for specifying video coding standards in the most natural manner. This is due to the fact that video coding is a data-oriented application and can easily be visualized and specified as a dataflow graph where actors correspond to functional units such as discrete cosine transform (DCT) and motion compensation (MC), and video bit stream flows among them via FIFOs. In addition to providing a natural language for specifying video codecs RVC-CAL provides a compilation framework called ORCC[1]. ORCC is implemented as an eclipse plug-in and has backends for C, VHDL and others and comes with various example applications including MPEG4 and HEVC decoders.

An RVC-CAL dataflow program is described as a set of interconnected actors via unidirectional FIFOs. Actors are composed of a set of actions, I/O ports (FIFOs) and internal state variables. Actors perform computation by firing actions depending on the state of their I/O ports and their internal state variables.

A multicore platform ideally runs an RVC-CAL program

[1] see http://orcc.sourceforge.net/

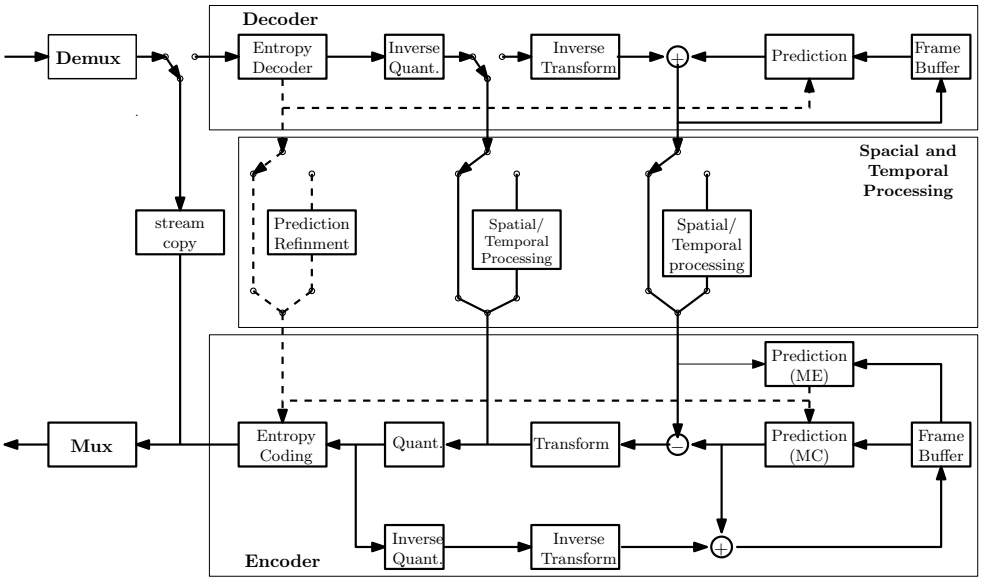

Fig. 2. Complete picture of video transcoding showing the main blocks it constitutes and the different options of doing transcoding depending on the required type of conversion operation, acceptable quality and complexity

by executing each actor in parallel as long as they are fireable. However because the number of actors in a typical dataflow application is usually much greater than the number of processors, several actors are usually executed on the same processor. The RVC-CAL runtime therefore maps and schedules actors using different approaches. Among others mapping is done in a round-robin or in a weighted load balancing shame, and scheduling is usually implemented using a round-robin or a data driven algorithm[11], [12].

III. Related Work

RVC-CAL [3], [4] uses CAL procedures to interface with legacy code and more specifically to call the I/O imperative functions such as file readers and display functions. CAL procedures are included in the RVC-CAL language for this particular purpose and can modify state variables and parameters passed to them by reference. Even though CAL procedures defeat some purposes of the dataflow programming approach such as side effect freeness they are, however practical considering the Von-Neumann architecture. In our work we propose an interface which can be used to call (use) dataflow components from the legacy code. This is important for legacy applications to tap in to dataflow components when appropriate.

In [13], Mark Green et al. have shown two ways of interfacing haskell which is a functional programming language and java an imperative language. Their motivation for their work was to provide each language an access to certain missing language features from the other. As a use case they showed how Java's I/O and UI related libraries which require imperative features that violate the referential transparency of functional programs can be used in haskell. On the other side they have also shown how the haskell features such as lazy evaluation and higher-order functions can be used in Java programs. In our work we explore a similar but more generic interfacing approach on a dataflow language (RVC-CAL) and the C imperative language using a more realistic application.

In [14], Chatterjee et al. presented the HCMPI programming model and runtime for programming distributed systems, which unifies asynchronous task parallelism at intra-node level with MPIs message passing model at the inter-node level. With HCMPI's task parallel model, users can benefit from MPI integration with structured task parallelism and dataflow programming. Similarly, in [15] and other research works around hybrid programming approaches, combining programming models is proven to be useful in dealing with hierarchical and various hardware designs.

In this paper, we propose the use of dataflow programming to express the available concurrency in video transcoding applications in a more natural way. At the same time, we allow to keep the effective legacy components of current transcoding applications. We enable this through the use of a generic interface defined between components implemented using different programming paradigms.

IV. Interfacing Dataflow with Legacy Code

Ideally we would need to find the most transparent way to accomplish the interfacing. It should be done in such a way that enables code in both languages to remain natural. There are two possible approaches to interface imperative code with dataflow code. One possible approach, adopted by current RVC-CAL application developments, is based on the interface driven by the dataflow code. The I/O imperative functions are called from the dataflow program using CAL procedures which are ad-hoc mechanisms put in place to accomodate legacy code in to dataflow components.

The approach proposed in this paper is also based on the use of imperative code for I/O. However in this approach it is the imperative code that calls the dataflow code with an input data to be processed. The main advantage of this approach is the ease of development. Each language is used to implement those parts of code for which the language is most appropriate for, without the need to accommodate the languages to each other (as in RVC-CAL procedures). This

enables independent prototyping and reuse of code already written. It also permits legacy libraries to access dataflow components and helps adoption of dataflow programing.

A. Interface Definition

The interface is designed to be generic enough such that any new dataflow component, like a decoder, encoder or filter, can be added to any legacy video processing library. The proposed interface consists of three functions and a data structure which are explained and implemented as follows.

1) init_component: This interfacing function is used to launch the dataflow component. Launching a dataflow component can be conceived as starting a conveyor belt system in a factory. Once a conveyor belt along with the processing units connected to it are started, a factory is ready to receive a stream of items to be processed. In our case, the initialization function starts the dataflow runtime system. The RVC-CAL runtime mainly consists of mapping, scheduling and other utility routines. The runtime takes in a set of actors, their network and user-supplied parameters such as an input data and maps, schedules and executes actors on a number of processing units. Mapping is either done by simply assigning actors to available processing units in a round-robin manner or via more complex post-profiling weight-based methods [11]. Once actors are mapped to a processing unit they are scheduled using round-robin or more advanced data-driven methods.[12].

Data: context
Result: success
1 set_component_context($context$);
2 success =
thread_create($launcher, launch, context, tid$);
Algorithm 1: Dataflow component initialization

The pseudo-code in Algorithm 1 shows the implementation of the *init_component* function. $context$ is any information that is needed to start the component. It contains the runtime options of the dataflow component such as mapping policy, scheduling policy, number of cores to be used and the dataflow network itself. Once the desired $context$ of the component is set, the dataflow component is launched with a new thread which ensures the *init_component* function returns control to the legacy code immediately. This allows the caller to continue its execution by sending row data and receiving processed data from the dataflow component.

2) process: This function is responsible for feeding the already initialized dataflow component with input data and grubbing the output data if available. Every call to this function from the legacy transcoder might fill the input FIFO of the dataflow component or get tokens from the output FIFO of the component, or both.

As shown in Algorithm 2, the *process* function takes in the *context* of the dataflow component which contains the network information and the data to be processed in *ipkt*. It then returns any result that might be available from the dataflow component in *opkt*. Note that this function also returns the size of the data consumed by the component through the variable *sent*. Any unconsumed data on the input FIFO of the dataflow component should be re-supplied to this function. *got_result* is used to tell a calling function if the dataflow component resulted in a valid output via *opkt*.

Data: context, ipkt
Result: sent, opkt, got_result
1 tosend=ipkt.size;
2 sent = 0;
/* send input to dataflow */
3 sent += send(context,ipkt, tosend);
4 last_processed=processed;
/* recive processed data */
5 processed += recv(context,opkt);
6 **if** $last\_processed < processed$ **then**
/* we have got data */
7 got_result = 1;
8 **else**
9 got_result = 0;
10 **end**
11 ipkt.size -= sent;
12 ipkt.data += sent;
Algorithm 2: Dataflow processing

3) close_component: This interface function is used to end the already running dataflow component. More specifically it ends the runtime of the component by joining all created threads that were responsible for executing the component's actors.

Data: context
Result: success
1 thread_join(launcher);
2 success = free_component_context(context);
Algorithm 3: Dataflow component termination

Algorithm 3 shows the *close_component* function.

4) Component Structure: This structure definition enables the use of multiple dataflow components and ensures the generic nature of the interface.

```
1 typedef struct component {
2     const char name;
3     enum type component_type;
4     enum id component_id;
5     struct component *next;
6     int (*init_component)(context *);
7     int (*process)(context *, opkt*, ipkt*, *got_result);
8     int (*close_component)(context *);
9 }
```

Algorithm 4: Component Structure

As can be noted from Algorithm 4, the component definition allows multiple dataflow components to be identified by a name or id. It also contains pointer to the three functions that are used to initiate, use and close a given dataflow component.

B. Generating Interface and Dataflow Component

In order to have an automated workflow for integrating dataflow components into a transcoding framework, we propose generating the interface automatically from the ORCC backend.

As shown in figure 3 we have modified the ORCC C backend to generate the dataflow components as a library along with a header file instead of stand alone executable. In addition

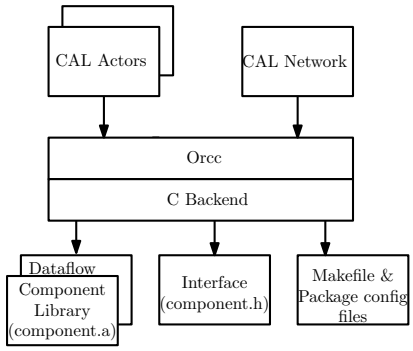

Fig. 3. Generation of the dataflow component library and its interface

we have added generation of package configuration files so that the library can be installed and be used easily.

C. Using the Interface

In order to demonstrate the functionality of the approach we have generated MPEG and HEVC video decoders form the corresponding dataflow descriptions, written in RVC-CAL, using our modified C backend. Figure 4 shows the structure of the HEVC dataflow decoder.

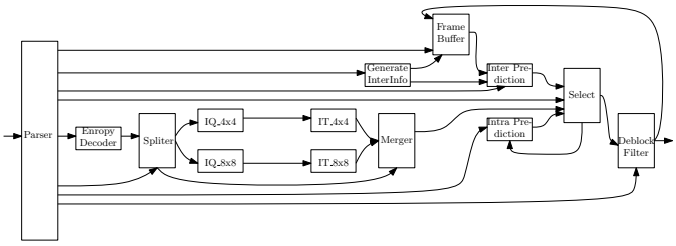

Fig. 4. Dataflow Decoder Implementation from ORCC

The generated dataflow component libraries are then compiled and installed in our system. Following that we have included the header files of the generated dataflow components in our legacy video transcoding framework and linked to the installed component libraries during its compilation.

Algorithm 5 shows the use of a dataflow decoder component by the legacy video transcoder FFmpeg. Three main points can be noted from the pseudo-code. The first is the `register_all` function that is used to register the available formats, codec and filters in a given system. This function is from FFmpeg libraries and we have also used it to register our new dataflow decoder component. The second point to note is the use of our interface which constitutes the three functions, *init_component, process and close_component*. This interface can be used to abstract the various types of dataflow components that can be implemented and integrated to FFmpeg or any other legacy transcoder. Finally one can note that the FFmpeg libraries supply the I/O (read/write) functions which are capable of parsing almost any known video container format efficiently. In addition to the I/O functions, data processing functionalities such as video scalers and encoders that are yet to be implemented by dataflow approach can also be used.

Using our interface we were therefore able to provide FFmpeg, a legacy video transcoding framework with dataflow components that implement fine-grained parallelism. Note from figure 4 that the dataflow decoder component provides a fine grained parallelism by implementing functional blocks as separate actors.

Data: v_s $(source\ video), context$
Result: v_p $(processed\ video)$

```
1  register_all();
2  init_component(context) ;          // initialize
   component e.g. decoder/filter/encoder
3  while read(context, v_s, ipkt) do
4  |  process(context, opkt, ipkt, got_result) ;  // e.g.
   |  decode/filter/encoder
5  |  if got_result then
6  |  |  rescale_frame (context, fpkt, opkt, got_result);
7  |  end
8  |  if got_result then
9  |  |  encode_frame (context, opkt, fpkt, got_result);
10 |  end
11 |  if got_result then
12 |  |  write(context, opkt);
13 |  end
14 |  ipkt.data += ret;
15 |  ipkt.size -= ret;
16 end
17 flush();
18 close_component(context) ;  // close component
   e.g. decoder/filter/encoder
```

Algorithm 5: FFmpeg overview

V. Experiments

In order to check the proper operation and benefits of our integration approach[2] we have made two evaluations which include quality and scalability measurements. In all of our experiments we used four different input videos from different categories with resolution of 1080p, bitrate of 1200kbps and frame rate of either 24 or 30 or 50. The transcoding operation performed in the experiments are resolution reduction transcoding. See result tables I and II for details on the input video characteristics and the transcoding parameters.

A. Quality

First, we made a quality difference measure between video transcoding operations which use the dataflow decoder and the original legacy decoders that comes with the FFmpeg transcoder. The original videos were of 1080p resolution and were transcoded to 240p, 480p or 720p. The quality difference were measured using three metrics, *psnr,ssim, msssim* [16], [17]. The visual similarity matrices *ssim, msssim* are close to 1.00 and the structural similarity metrics *psnr* is *inf* for all tests indicating that the resulting videos are similar. This means our proposed interfacing works properly.

B. Evaluation of Scalability

Besides showing the proposed interface works correctly we here present scalability measurements on a 6 core Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00GHz Machine. Table I shows the speed up of 62-80% when using 4 cores for a dataflow component (MPEG4 part 2 simple profile decoder) against a single core. Similarly Table II shows the speed up of 48-65%

[2]https://github.com/tdeneke/ffmpeg-2.5.3, https://github.com/tdeneke/orcc and https://github.com/tdeneke/orc-apps

when using 4 cores for the dataflow component (HEVC simple profile decoder) against using a single core. The remaining 2 cores were assigned to the rest of the transcoding framework components such as the encoder. The speed up is calculated as $\frac{(fps_4 - fps_1)}{fps_1} * 100$ where fps_4 is the transcoding speed in frames per second (fps) when using 4 cores for the dataflow component and fps_1 is the transcoding speed in fps when using 1 core for the dataflow component. Each mesurment is repeated 10 times to calculate the confidence intervals.

TABLE I. EXPERIMENTAL RESULTS SHOWING THE FRAME RATE (FPS) OF THE DIFFERENT VIDEO STREAMS AND THE CONFIDENCE INTERVAL FOR A 97.5% CONFIDENCE LEVEL. THE SPEEDUP FROM RUNNING THE DATAFLOW COMPONENT OF THE TRANSCODER WITH A SINGLE CORE TO RUNNING IT IN FOUR CORES IS GIVEN FOR EACH RESOLUTION AND VIDEO SEQUENCE.

	240p	480p	720p
Cartoon - Elephant Dreams - 24fps			
1	3.456 ± 0.036	3.319 ± 0.027	3.075 ± 0.039
2	4.808 ± 0.011	4.529 ± 0.051	4.092 ± 0.027
4	6.085 ± 0.172	5.422 ± 0.537	4.995 ± 0.074
speedup	76.07%	63.36%	62.44%
Consumer Video - Old Town Cross - 50fps			
1	3.432 ± 0.030	3.302 ± 0.029	3.065 ± 0.036
2	4.805 ± 0.012	4.535 ± 0.017	4.106 ± 0.011
4	6.180 ± 0.066	5.742 ± 0.050	4.996 ± 0.172
speedup	80.07%	73.89%	63.00%
Documentary - Snow mountain - 30 fps			
1	3.487 ± 0.010	3.386 ± 0.040	3.191 ± 0.025
2	4.822 ± 0.028	4.607 ± 0.047	4.223 ± 0.020
G4	6.064 ± 0.380	5.863 ± 0.071	5.171 ± 0.292
speedup	73.90%	73.15%	62.05%
Sport - Touchdown Pass - 30 fps			
1	3.373 ± 0.040	3.241 ± 0.031	2.975 ± 0.033
2	4.528 ± 0.382	4.383 ± 0.073	3.917 ± 0.022
4	5.508 ± 0.764	5.478 ± 0.250	4.519 ± 0.583
speedup	63.3%	69.02%	51.9%

TABLE II. EXPERIMENTAL RESULTS SHOWING THE FRAME RATE (FPS) OF THE DIFFERENT VIDEO STREAMS AND THE CONFIDENCE INTERVAL FOR A 97.5% CONFIDENCE LEVEL. THE SPEEDUP FROM RUNNING THE DATAFLOW COMPONENT OF THE TRANSCODER WITH A SINGLE CORE TO RUNNING IT IN FOUR CORES IS GIVEN FOR EACH RESOLUTION AND VIDEO SEQUENCE.

	240p	480p	720p
Cartoon - Elephant Dreams - 24fps			
1	1.474 ± 0.001	1.473 ± 0.001	1.467 ± 0.002
2	1.800 ± 0.001	1.797 ± 0.003	1.791 ± 0.003
4	2.193 ± 0.089	2.185 ± 0.144	2.233 ± 0.033
speedup	48.78%	48.34%	52.22%
Consumer Video - Old Town Cross - 50fps			
1	1.396 ± 0.001	1.394 ± 0.002	1.391 ± 0.001
2	1.729 ± 0.003	1.729 ± 0.004	1.723 ± 0.003
4	2.176 ± 0.025	2.136 ± 0.119	2.144 ± 0.084
speedup	55.87%	53.23%	54.13%
Documentary - Snow mountain - 30fps			
1	1.493 ± 0.002	1.489 ± 0.003	1.482 ± 0.002
2	1.916 ± 0.011	1.916 ± 0.006	1.905 ± 0.001
4	2.461 ± 0.065	2.460 ± 0.044	2.273 ± 0.164
speedup	64.84%	65.21%	53.37%
Sport - Touchdown Pass - 30fps			
1	1.307 ± 0.001	1.304 ± 0.001	1.299 ± 0.001
2	1.356 ± 0.673	1.595 ± 0.002	1.588 ± 0.003
4	1.993 ± 0.054	2.009 ± 0.023	2.004 ± 0.009
speedup	52.49%	54.06%	54.27%

VI. CONCLUSION

In this paper we have proposed the use of a generic interface for integrating dataflow components such as decoders, encoders and filters with legacy transcoding libraries. The interface enables seamless interaction between dataflow and legacy imperative code allowing each programming approach to implement components for which it is appropriate for.

We have also tested and shown the proper functionality of our approach. Scalability evaluations also show the gain that can be obtained from using dataflow components via the proposed interface.

In the future we would like to further explore the effect of the dataflow component runtime and transcoding framework runtime on each other.

REFERENCES

[1] E. A. Lee, "The problem with threads," *Computer*, pp. 33–42, May 2006.

[2] Orcc, "Open RVC-CAL compiler," 2009. [Online]. Available: http://orcc.sourceforge.net/

[3] M. Wipliez, "Compilation infrastructure for dataow programs," Ph.D. dissertation, INSA Rennes, Sep. 2010.

[4] H. Yviquel, "From dataflow-based video coding tools to dedicated embedded," Ph.D. dissertation, UNIVERSITE DE RENNES 1, Oct. 2013.

[5] S. F. Chang and A. Vetro, "Video adaptation: Concepts, technologies, and open issues," *Proceedings of IEEE*, Jan 2005.

[6] J. Xin, C.-W. Lin, and M.-T. Sun, "Digital video transcoding," *Proceedings of the IEEE*, Jan. 2005.

[7] ffmpeg, "ffmpeg," 2000. [Online]. Available: https://www.ffmpeg.org/

[8] E. A. Lee and T. M. Parks, "Readings in hardware/software co-design," Norwell, MA, USA, 2002, pp. 59–85.

[9] G. Kahn, "The semantics of a simple language for parallel programming," in *Information processing*, J. L. Rosenfeld, Ed. Stockholm, Sweden: North Holland, Amsterdam, Aug. 1974, pp. 471–475.

[10] J. Eker and J. W. Janneck, "Cal language report: Specification of the cal actor language," University of California, Berkeley, Berkeley, California, USA, Tech. Rep., 2003.

[11] H. Yviquel, E. Casseau, M. Raulet, P. Jaaskelainen, and J. Takala, "Towards run-time actor mapping of dynamic dataflow programs onto multi-core platforms," Sept 2013, pp. 732–737.

[12] H. Yviquel, E. Casseau, M. Wipliez, and M. Raulet, "Efficient multicore scheduling of dataflow process networks," in *SiPS*, Oct 2011, pp. 198–203.

[13] M. Green and A. E. Abdallah, "Interfacing java with haskell." in *Scottish Functional Programming Workshop*, ser. Trends in Functional Programming, vol. 1, 1999, pp. 79–88.

[14] S. Chatterjee, S. Tasrlar, Z. Budimlic, V. Cave, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan, "Integrating asynchronous task parallelism with mpi," in *IPDPS*, May 2013, pp. 712–725.

[15] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes," Feb 2009, pp. 427–436.

[16] Z. Wang, A. Bovik, H. Sheikh, and E. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *Image Processing, IEEE Transactions on*, 2004.

[17] vqmt, "vqmt," 2013. [Online]. Available: http://mmspg.epfl.ch/vqmt