EFFICIENT SOFTWARE SYNTHESIS OF DYNAMIC DATAFLOW PROGRAMS

H. Yviquel[⋆] *A. Sanchez*[⋆] *P. Jääskeläinen*[‡] *J. Takala*[‡] *M. Raulet*[⋆] *E. Casseau*[†]

[⋆] INSA of Rennes, IETR, France.
[‡] Tampere University of Technology, Finland.
[†] University of Rennes 1, IRISA, Inria, France.

ABSTRACT

This paper introduces advanced software synthesis techniques that enhance the implementation of dynamic dataflow programs. These techniques have 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. The results show an improvement of more than 100% of the frame-rate over previously proposed implementations, and achieve real-time decoding of high definition video sequences.

1. INTRODUCTION

The emergence of massively parallel architectures, along with the increasing complexity of applications, has revived the interest in dynamic dataflow programming. Indeed, dynamic dataflow programming offers a flexible development approach which is able to build complex and modular applications while expressing parallelism explicitly. Paradoxically, most of the studies stay focused on static dataflow programming, even if a pragmatic development process requires the expressiveness and the practicality offered by dynamic dataflow programming.

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. While the efficiency of traditional language programs is 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.

As a result, this paper presents advanced software synthesis techniques that enhance the implementation of dynamic dataflow programs using their specific properties and the flexibility of software systems. These techniques have 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.

The paper is organized as follows. First, the context of dynamic dataflow programming is described in Section 2. Then, we describe our methodology to enhance the software synthesis of dynamic dataflow programs in Section 3. Section 4 presents experimental results and compare them with previous works. Finally, we conclude in Section 5.

2. DYNAMIC DATAFLOW PROGRAMMING

Dynamic dataflow programming relies upon a model of computation called Dataflow Process Network (DPN) [1], which is closely related to Kahn Process Network (KPN). In this model, an application is represented as a directed graph wherein the vertices model computational units that are called *actors* and the unidirectional edges represent unbounded communication channels based on FIFO principle. The FIFO channels can be empty or can carry a possibly infinite sequence of 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 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 which specifies when the functions can be fired, i.e. the number and the values of tokens that have to be available on the input ports to fire the actor. An actor can fire when at least one of its firing rules is satisfied. When several firing rules are satisfied at the same time, a single one is chosen based on predefined priorities.

Few years ago, MPEG has introduced an innovative framework, called RVC [2, 3], 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,

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).

978-1-4799-2893-4/14/$31.00 ©2014 IEEE

in a modular and reusable fashion thanks to the inclusion of a subset of CAL programming language [4], and the support of a complete development environment known as Orcc [5].

In general, communication and synchronization are the major sources of inefficiencies on every multi-core system. In particular, the implementation of dynamic dataflow programs faces two issues to achieve performance requirements: **Scheduling** and **communication**. Both are directly impacted by the application granularity, usually defined as the ratio of computation to the amount of communication. Video decoders are traditionally described at fine-granularity since the pixels are processed block after block. On the one hand, the scheduling is a well-known bottleneck of dynamic dataflow programs since their expressive power requires a large number of control structures. The literature has already introduced a large panel of methodologies to optimize the scheduling of dynamic dataflow programs in different manners [6, 7, 8, 9, 10, 11]. On the other hand, the communication is the major bottleneck of all dataflow programs. Since the actors can only communicate through the FIFO channels, the execution requires a massive amount of data movements that can ultimately lead to poor performance. Restricted dataflow models usually solve this issue by grouping the data transfers, but this is not possible with dynamic dataflow models. As a result, this paper focuses on communication and computation aspects to enhance the software implementation of dynamic dataflow programs [12, 13] using the specific properties of the DPN model and the flexibility of software systems.

3. PROPOSED SOFTWARE IMPLEMENTATION OF DYNAMIC DATAFLOW PROGRAMS

In theory, the DPN model defines FIFO channels with unbounded capacity [1]. 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.

3.1. 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.

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.

```
1  transp: action
2   IN:[ src ] repeat 16 // Input pattern
3    ==>
4   OUT:[ dst ] repeat 16 // Output pattern
5  var
6    int(size=16) dst[16] =
7      [ src[ 4 * column + row ] :
8        for int row in 0 .. 3,
9         for int column in 0 .. 3
10     ]
11 end
```

Listing 1. Transposition of a 4x4 block in CAL

3.2. Copy-Free Communications

One of the high-level features of CAL is its ability to describe *multi-rate* actions [4], i.e. actions reading and writing pools of data at each firing, such as the very simple example presented in Listing 1, 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 1, is translated into a function composed of 3 steps as follows [14, 12]: 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 1, 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 1, 16 tokens are writ-

ten 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 bIn fact, the firing rules are evaluated successively according to the *partial order* defined within the actor (priorities and FSM). Thus,uffers and the local variables (knowing that only one copy is mandatory).

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

Listing 2. 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 output 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 [1], 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 2. 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.

3.3. 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 2) and aligned (Listing 3), 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.

```
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 3. Aligned action

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 (similarly than 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.

3.4. Multi-level Dynamic Scheduling

As defined by Lee and Parks [1], 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* [13] such as presented in Listing 4. 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`).

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 previous works [15, 16], 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.

```
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 4. Action scheduler

To conclude, the execution of DPN-based programs involves a complex scheduling that have to be performed at run-time. 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.

4. RESULTS

This section studies the implementation of dynamic dataflow programs on both desktop and embedded multi-core platforms. On the one hand, the desktop implementation is generated by use of the C back-end of Orcc [5]. The generated C code is compiled with GCC and executed on top of Ubuntu GNU/Linux. Concerning the platform that has been used during these experiments, we use an Intel Core i7 with 2 cores clocked at 2.8GHz. On the other hand, the embedded implementation targets multi-core platforms composed of homogeneous Very Long Instruction Word -style processors, based on the Transport-Trigger Architecture (TTA) [17], running at 100MHz and interconnected by point-to-point shared memories. In this configuration, the tested software implementations are generated by use of the TTA back-end of Orcc [18], then the generated code is compiled and simulated thanks to the TTA-based Co-design Environment (TCE) [19].

	Desktop			**Embedded**	
	[20]	[13]	Ours	[18]	Ours
MPEG-4 SP	12	150	400	90	180
MPEG-4 AVC	N/A	60	220	N/A	N/A

Table 1. Improvement of more than 100% of the decoding frame-rates (QCIF) over previously proposed implementations [20, 13, 18]

Table 1 summarizes the decoding frame-rates obtained from different implementations of DPN-based video decoders. All the results have been obtained with the same application descriptions (standardized) and video sequences (*foreman* QCIF). The results clearly show that our implementation significantly improves the performance over previously proposed implementations thanks to our advanced software synthesis techniques. The results show an improvement of more than 200% on desktop platforms and of about 100 % on embedded platforms.

Number of cores	1	2	4
MPEG-4 SP	43	76	116
HEVC	13	20	25

Table 2. Real-time HD decoding frame-rates (720P) on desktop multi-core platforms

Table 2 presents the decoding frame-rates of HD video sequences obtained from our implementation of MPEG-4 SP (*Old town cross* 720P at 6Mbps) and HEVC (*Kristen And Sara* 720P at 1Mbps) on desktop multi-core platforms. The results show that our implementation can already achieve real-time decoding frame-rates even on a small number of processor cores and without assembly-level optimizations. The HEVC description is still being developed which explains the large difference of performance with MPEG-4 SP.

5. CONCLUSION

We have proposed advanced software synthesis techniques to enhance the implementation of dynamic dataflow programs on both desktop and embedded processors. We have particularly focused on communication and computation issues that are tackled using branch-free, copy-free and aligned implementations. Our implementation have shown an improvement of more than 100% of the decoding frame-rates over previously proposed implementations. Additionally, our approach has been validated by presenting real-time performance on HD video sequences using MPEG-4 SP and HEVC decoders.

6. REFERENCES

[1] Edward A. Lee and Thomas Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, 1995.

[2] Marco Mattavelli, Mickaël Raulet, and Jörn W. Janneck, "MPEG reconfigurable video coding," in *Handbook of Signal Processing Systems*, Shuvra S. Bhattacharyya, Ed F. Deprettere, Rainer Leupers, and Jarmo Takala, Eds., pp. 281–314. Springer, New York, NY, USA, 2013.

[3] Marco Mattavelli, Ihab Amer, and Mickaël Raulet, "The Reconfigurable Video Coding Standard [Standards in a Nutshell]," *Signal Processing Magazine, IEEE*, vol. 27, no. 3, pp. 159–167, 2010.

[4] Johan Eker and Jörn W. Janneck, "CAL language report: Specification of the CAL actor language," Tech. Rep., University of California, Berkeley, Berkeley, 2003.

[5] 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 conference on Multimedia*, 2013.

[6] Jani Boutellier, Mickaël Raulet, and Olli Silvén, "Automatic Hierarchical Discovery of Quasi-Static Schedules of RVC- CAL Dataflow Programs," *Journal of Signal Processing Systems*, vol. 71, no. 1, pp. 35–40, 2013.

[7] 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.*, 2012, pp. 1661–1664.

[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*, 2012, pp. 1465–1469.

[9] Jérôme Gorin, Mickaël Raulet, and Françoise Prêteux, "Optimized dynamic compilation of dataflow representations for multimedia applications," *Annals of telecommunications - Annales des télécommunications*, vol. 68, no. 3-4, pp. 133–151, 2012.

[10] Matthieu Wipliez and Mickaël Raulet, "Classification and transformation of dynamic dataflow programs," in *Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on*, 2010.

[11] Ruirui Gu, Jörn W. Janneck, Mickaël Raulet, and Shuvra S. Bhattacharyya, "Exploiting Statically Schedulable Regions in Dataflow Programs," *Journal of Signal Processing Systems*, vol. 63, no. 1, pp. 129–142, Jan. 2010.

[12] Matthieu Wipliez, Ghislain Roquier, and Jean-François Nezan, "Software Code Generation for the RVC-CAL Language," *Journal of Signal Processing Systems*, vol. 63, no. 2, pp. 203–213, June 2009.

[13] Matthieu Wipliez, *Compilation infrastructure for dataflow programs*, Ph.D. thesis, INSA of Rennes, Dec. 2010.

[14] 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*, 2008, pp. 281–286.

[15] 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*, 2011, pp. 198–203.

[16] Hervé Yviquel, Emmanuel Casseau, Mickaël Raulet, Pekka Jääskeläinen, and Jarmo Takala, "Towards runtime actor mapping of dynamic dataflow programs onto multi-core platforms," in *Image and Signal Processing and Analysis (ISPA), 2013 8th International Symposium on*, 2013.

[17] Henk Corporaal, *Microprocessor Architectures: from VLIW to TTA*, John Wiley & Sons, Chichester, UK, 1997.

[18] 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, Special issue on Reconfigurable Video Coding*, 2013.

[19] 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*, 2010, pp. 217–222.

[20] Anders Carlsson, Johan Eker, Thomas Olsson, and Carl Von Platen, "Scalable parallelism using dataflow programming," *Ericsson Review*, vol. 2, no. 1, pp. 16–21, 2010.