STATIC AND QUASI-STATIC COMPOSITIONS OF STREAM PROCESSING APPLICATIONS FROM DYNAMIC DATAFLOW PROGRAMS

Johan Ersfolk[1,3]*, Ghislain Roquier*[2]*, Wictor Lund*[1]*, Marco Mattavelli*[2]*, Johan Lilius*[1]

[1]Åbo Akademi University, Finland
[2]École Polytechnique Fédérale de Lausanne, Switzerland
[3]Turku Centre for Computer Science, Finland

ABSTRACT

Dynamic dataflow models for their expressiveness properties have shown to represent more adequate and attractive solutions for describing state of the art signal processing applications. However, they are known to present potential run-time penalties when implementations are obtained by mapping and scheduling a dataflow network partition on a processing unit. In general terms, a completely static scheduling at compile-time of dynamic dataflow programs remains an unsolved problem. Several approaches for the composition of actors are promising approach that can significantly reduce the potential penalty of run-time scheduling thus increasing the overall performance of the system. This paper presents static and quasi-static composition techniques that results in a reduction of the portion of dynamic dataflow networks, by applying appropriate transformations to network partitions that after a specific analysis demonstrate to possess a predictable behaviour. Some experiments based on a video processing application ported on several system-on-chips show the achievable speedup corresponding to the reduction of the number of run-time scheduling decisions.

Index Terms— Dataflow Process Network, Actor Composition, Static and Quasi-Static Scheduling

1. INTRODUCTION

Dataflow programming models have a rich history dating back to at least the early 1970s, including seminal work by Dennis [1] and Kahn [2]. For the purpose of this work, a *dataflow program* is a directed graph in which nodes represent computational units (called *actors*), while edges represent connections between actors used to communicate sequences of data packets (*tokens*). Several variations of this kind of dataflow model have been introduced in literature [2, 3, 4, 5], often referred to as different dataflow *models of computation* (MoC). They differ in the kind of actor behaviour and they permit to use different techniques for the scheduling of the actor execution, which results in different trade-offs between expressiveness and implementation efficiency. One common property across all these dataflow models is that individual actors encapsulate their own state, and thus do not share memory with one another. Instead, actors communicate with each other exclusively by sending and receiving tokens along the channels connecting them. The resulting absence of race conditions makes the behaviour of dataflow programs more robust to different execution policies, whether those be truly parallel or some interleaving of the individual actors.

Dynamic behaviors appears in most of stream processing applications since some decisions must be done at run-time when the operations to be performed on the data stream depend on the information carried by the data itself. For instance, deciding whether an intra type or inter type of video processing has to be applied to incoming data is an example of such kind of run-time decisions that can not be avoided. Dynamic dataflow models are sufficiently expressive to naturally model such kind of applications and this property is one of the main reasons for which they are widely used. However, dynamic dataflow requires a dynamic scheduler, that takes decision at run-time to schedule the appropriate execution of actor. Hence, a significant unnecessary overhead can be potentially added during execution.

In the paper, several techniques capable of reducing the dynamism of dataflow program executions, based on static analysis and abstract interpretation of dataflow networks are presented. Those techniques are applied to an MPEG-4 Part2 decoder implemented on different system-on-chips platforms that show the speedup achievable when reducing the number of unnecessary run-time decisions.

2. PROCESS NETWORKS

The developments and results presented in this paper are related to the dataflow variant of *Kahn process network* (KPN) [2] named *dataflow process network* (DPN) [5]. KPN is a well-known model widely used for describing digital signal processing systems. By contrast with KPN, where processes (actors) represent continuous mapping from input sequences to output sequences, DPN specifies processes as a sequences of discrete computational steps, called *firings*. In each of such step, an actor may consume a finite number

of input tokens, produce a finite number of output tokens, and modify its internal state, if it has any. The behaviour of a DPN actor is specified as a pair of *firing rule* and *firing*. The *firing rule* determines when the actor may fire, by describing the input sequences and state that need to be present for the actor to be able to make a step, i.e. for it to be *enabled*. The *firing* determines, for each input a sequence/state combination for which the actor is enabled according to the *firing rule*, the output tokens produced in that step and, if applicable, the new actor state. In general, an actor may be non-deterministic, which means that the firing function may yield more than one combination of output and next state for any given *enabled* actor.

The CAL actor language [6] is a formal programming language capable of implementing DPN actors semantic. Each actor is defined by a set of *actions* where each action captures a part of the firing rule along with the part of the firing that pertains to the input/state combinations enabled by that partial rule. An action is enabled according to its *input patterns* and *guard* expressions. Input patterns define the amount of data that are required on the input sequences, whereas guards are boolean expressions on the current state and/or on input sequences that need to be satisfied for enabling the execution of an action.

3. ACTOR COMPOSITION: THE SCHEDULING PROBLEM

Various compilers for Cal language targeting parallel platforms (multicore, many-core and FPGA) are available and have shown to be well adapted to generate implementations for massively parallel platforms such as FPGAs [7]. In such scenario, the degree of parallelism is assumed to be very high and a one-to-one mapping of each individual actor is performed, hence a computational unit is created for each actor. However, one-to-one mapping from actors to processing units is neither possible nor appropriate in most of the cases. In other words, the potential parallelism of applications is in general much higher than the available parallelism provided by platforms, therefore actors need to share the same processing unit. For this reason, a scheduler is responsible to decide at run-time which actor to fire at each step and the firing sequence can only be determined dynamically. However, some DPN actors may belong to more restricted MoCs. Examples of more restrictive dataflow MoCs include synchronous dataflow (SDF) [3], cycle-static dataflow (CSDF) [4], where the scheduling decision can be done at compile-time. Knowing that, it is possible to reduce the dynamic scheduling overhead introduced by DPN by gathering together actors that belong to more restricted MoCs. This operation is known as actor composition. Actor composition is the process of transforming a network of actors that belong to a given MoC into a composite actor, without affecting the global processing semantics.

Statically schedulable MoCs (static MoCs for short) have the nice property that they have minimal run-time overhead as all scheduling decisions can be done at compile-time [3]. SDF and CSDF are two well-known classes of static MoCs. SDF is a special case of DPN where firing rules have the same tokens rate. CSDF relaxes the SDF in that it can have firing rules with different token rates, with the constraint that its inner firing sequence is cyclic. Static actors (SDF and CSDF) have predictable behaviour, the scheduler executes repeatedly the firing sequence, hence it does not have to check any conditions at runtime. SDF is not as expressive as DPN and may not be able to describe some processing behaviours needed in common signal processing applications. For example a video decoder needs to decode several different types of blocks, each requiring a slightly different set of computations and schedules. The schedule selection can only be done at run-time based on information carried by the video data stream itself.

For actors that are not statically schedulable, quasi-static (or piecewise static) scheduling methods can be used to derive schedules for actor composition. With quasi-static scheduling, the idea is to make most of the scheduling decisions at compile-time while leaving only some necessary scheduling decision for run-time. The key conditions that make that possible must be identified while other redundant conditions are removed to create static schedules. The result is a set of conditions associated to static schedules (lists of action firings) and in some cases a state machine to describe possible scheduling states of the composed actor. Quasi-static scheduling is typically needed when the scheduling of actors does not strictly depend on token rates but also on the value of input tokens and state variables. Such actors are not by them selves statically schedulable as most of the actors are guarded by, from the actors point of view, unknown information. Another situation is when it is not possible to complete one actor at the time but the execution of the actors needs to be interleaved to complete. Interleaving is typically needed when there are feedback loops in the dataflow network. The idea pursued here is that when a network partition including several actors is analyzed as a whole, many of the checks required to execute a single actor becomes redundant.

4. RELATED WORKS AND THE PROPOSED SCHEDULING APPROACH

In the paper, we proposed two techniques for the static and the quasi-static scheduling of dynamic dataflow programs. There is an abundant literature related to the compile-time scheduling of static dataflow MoCs. The challenge beyond that, is how to apply such techniques to dynamic MoC. Several approaches have been proposed to statically schedule CAL programs [8]. In our approach, a first assumption is that a classification of actors is first applied on the input dataflow network, hence actors are classified according to the MoCs they belong

to. We used the classifier proposed by [9, 10] to this end. Then, we took care about the fact that composition of static actors is not always possible, hence composing two connected actors may introduce deadlock. However, sufficient conditions that guarantees deadlock-freedom have been presented in [11]. For that purpose, we proposed a pairwise clustering method to compose static actors that construct the compile-time firing sequence, based on token rates information [3] and the composing conditions [11]. The process can be described as follow: an actor is randomly selected, if it has static MoC, every connected actor is tested for composition. An Actor is merged if it is also static and if the conditions are met, otherwise it is rejected. Note that the algorithm is greedy, hence it does not always output the optimal set of actors that are composable (by means of given objective metrics).

For the actors which are not static, quasi-static scheduling methods can be used to compose the actors. Some approaches for deriving quasi-static schedules for CAL programs have been proposed [12, 8, 13]. In general terms, these methods identify statically schedulable portions of data flow networks where the actors are not completely statically schedulable. Another approach proposed by Janneck in [14], uses a machine model to describe the actors such that these can be composed.

In the approach in [15, 13], a partition of actors is analyzed and the possible inputs (scenarios) are used to identify deterministic schedules that link recurring network execution states. Therefore, the only dynamic operation of the scheduler remains the guard evaluations between states linked by the obtained deterministic schedules. A model checker is used to find paths (schedules) between reachable states where a run-time scheduling decision is needed. The scheduler of a composed actor will simply choose a schedule, consisting of a sequence of actions, based on a condition which is derived from the guards of the original actors.

5. EXPERIMENTS

Figure 1(a) depicts a dataflow program implementing an MPEG-4 Part 2 decoder. It contains 13 actors that corresponds to the entropy decoding (syntax parser and the variable length decoder), residual decoding (AC-DC predictions, inverse scan, inverse quantization and IDCT) and the motion compensation (framebuffer, interpolation and residual error addition). The source (S) reads the bitstream while the sink (D) displays the decoded video. The color of actors represents the MoC they belong to, viz. SDF/CSDF in green, QSDF in yellow and DPN in red. The source and sink are classified manually as DPN since they are I/O actors that should not be composed.

Composition is shown in Fig. 1(b). The residual decoding includes 3 SDF and 1 CSDF actors. They are assembled into a composite CSDF actor (since SDF is a particular CSDF). On the other side, the motion compensation includes 2 QSDF and 1 SDF actors, hence a QSDF composite actor is inserted.

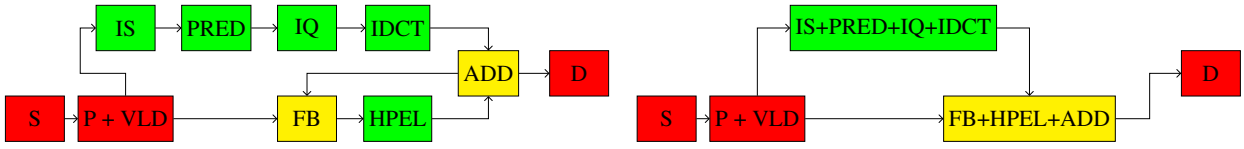
	QCIF	720p	1080p
Raspberry Pi ARM11 @ 800 MHz - minimal FIFOs			
orig	59.8 ± 0.1	1.85 ± 0.01	0.82 ± 0.01
qsdf	65.0 ± 0.1	2.16 ± 0.01	0.93 ± 0.01
sdf	65.7 ± 0.1	2.11 ± 0.01	0.91 ± 0.01
both	68.0 ± 0.1	2.24 ± 0.01	1.01 ± 0.01
speedup	13.7%	21.0%	23.2%
Raspberry Pi ARM11 @ 800 MHz - 4k FIFOs			
orig	62.9 ± 0.1	2.17 ± 0.02	0.94 ± 0.02
qsdf	85.7 ± 0.1	2.85 ± 0.02	1.24 ± 0.02
sdf	96.4 ± 0.1	3.19 ± 0.01	1.39 ± 0.02
both	92.2 ± 0.1	3.04 ± 0.02	1.39 ± 0.02
speedup	53.2%	40.0%	47.9%
Pandaboard ARM Cortex-A9 @ 1.2 GHz - minimal FIFOs			
orig	256 ± 1	8.38 ± 0.03	3.68 ± 0.05
qsdf	275 ± 1	9.58 ± 0.03	4.18 ± 0.06
sdf	259 ± 1	8.75 ± 0.03	3.77 ± 0.05
both	286 ± 1	9.86 ± 0.05	4.34 ± 0.06
speedup	11.7%	17.7%	17.9%
Pandaboard ARM Cortex-A9 @ 1.2 GHz - 4k FIFOs			
orig	167 ± 1	5.77 ± 0.03	2.49 ± 0.03
qsdf	190 ± 1	5.56 ± 0.03	2.84 ± 0.03
sdf	196 ± 1	6.70 ± 0.03	2.91 ± 0.03
both	199 ± 1	6.81 ± 0.03	2.96 ± 0.03
speedup	19.2%	18.0%	18.9%
Intel Xeon E5 2620 @ 2 GHz 4 - minimal FIFOs			
orig	1068 ± 1	35.4 ± 0.1	15.4 ± 0.2
qsdf	1183 ± 1	40.3 ± 0.1	17.7 ± 0.2
sdf	1090 ± 1	35.9 ± 0.1	15.7 ± 0.2
both	1233 ± 1	41.7 ± 0.1	18.1 ± 0.2
speedup	15.5%	17.8%	17.5%
Intel Xeon E5 2620 @ 2 GHz - 4k FIFOs			
orig	1328 ± 1	45.3 ± 0.1	19.8 ± 0.2
qsdf	1548 ± 1	51.8 ± 0.1	22.6 ± 0.3
sdf	1548 ± 1	50.8 ± 0.1	22.4 ± 0.2
both	1574 ± 1	52.5 ± 0.1	23.1 ± 0.3
speedup	18.5%	15.9%	16.7%

Table 1. Experimental Results showing the frame rate (fps) of the different configurations and the confidence interval for a 99% confidence level. The speedup from the original decoder to the fastest decoder with composed actors is given for each platform and video sequence.

5.1. Experimental Setup

The purpose of the experiments is to determine the impact of the static and quasi-static compositions on the performance of the overall video decoder, for this reason the experiments are run with 1) the original decoder, 2) the motion compensation network composed as QSDF, 3) the residual coding network composed as CSDF and 4) both networks composed as QSDF and CSDF respectively.

The experiments are run on three platforms with rather



(a) Original (b) With composition

Fig. 1. Dataflow description of the MPEG-4 part2 *Simple Profile*, for the sake of clarity the multiple channels between connected actors are not represented.

different properties regarding cache memories and instruction-sets. The results from different architectures are not interesting to be compared in absolute terms, instead we can compare the impact of different compositions on the architectures. For the set of decoders and platforms we use a set of videos of various size which affects the memory usage of the decoders by requiring different size of the frame buffer. The memory usage is further altered by running the decoders both with large FIFOs (4k) and minimal FIFOs sizes.

For the experiments, the three platforms run Linux (Raspbian and Ubuntu) and the code generated from the CAL program is compiled for the three platforms using gcc 4.6.3. The decoders are instrumented with code to collect statistics about the decoding progress. With a five second interval the decoder records the number of frames coded since the last interval, after running the decoder for several minutes when at least 30 readings has been collected, these are used to calculate the average frame rate and the confidence interval (for the average frame rate during five seconds) according to the standard normal distribution.

5.2. Experiment Results

The performance of the programs generated from CAL is affected by both the number of conditional branches in the scheduler and from how the schedules affect the cache behaviour. On the one hand, our goal is to have long static schedules, where a single condition enables a sequence of many action firings and by this minimizes the number of conditions the program scheduler must execute. At the same time, we do not want the (composed) actors to become too large for the instruction cache. On the other hand, large FIFOs, increase the number of times the same (composed) actor is executed in sequence, thus improving the instruction cache, but also increase the memory usage of the decoder.

For these reasons, the performance, depending on actor composition and scheduling, can be very different on different architectures. To get a more complete picture of the impact of actor composition on the performance of a CAL program, we run the experiments of three platforms with rather different memory and cache sizes and structure. On the Raspberry Pi the L2 cache is not used by the ARM core making the L1 cache the Last Level Cache (LLC), the Pandaboard has a 1 MB L2 cache as LLC and the Xeon has a 15 MB L3 cache as LLC. Also the instruction sets are different for the three processors, while we obviously for the Xeon used x86, the two ARM processors support different instruction sets; for the ARMv7 of the Pandaboard we used the more dense Thumb-2 instruction set while for the ARMv6 of the Raspberry Pi we used the ARM instruction set.

In the results in Table 1 we can see how the performance of the decoder, in this case with respect to the frame rate, is improved by the static and quasi-static scheduling of the residual coding and motion compensation respectively and the combination of these. The table shows that composition of actors sharing a processor in most cases improved the performance of the program, the static composition always improved performance and the quasi-static in every but one case where static composition alone performed better.

Another interesting detail is that, for the Pandaboard, the performance was degraded with large buffers while the other platforms experienced improved performance. A possible explanation for this is that, while each of the processors have equally large L1 caches, beyond L1 the Raspberry Pi has no cache and the Xeon has much larger cache, as a consequence the increased memory usage has a large impact on the Pandaboards L2 cache while on the other platforms the impact is minimal.

While eliminating run-time scheduling decisions improves the performance of the decoder, considering the platform, in the composition and scheduling of actors, would enable new possibilities to improve the performance of the dataflow program. A possible alternative to searching for as large as possible static or quasi-static compositions, would then be to search for compositions and schedules that makes the best use of the cache.

6. CONCLUSIONS

The paper presented methods for static and quasi-static composition of dynamic dataflow programs. Experiments show that the static and quasi-static composition techniques reduce the run-time overhead of the target application and consequently increase its run-time performance. Those methods used jointly can significantly improve the performance of dynamic dataflow programs, closing up the gap with existing optimized software implementations.

7. REFERENCES

[1] Jack Dennis, "First version of a data flow procedure language," in *Programming Symposium, Proceedings Colloque sur la Programmation Paris, April 9-11, 1974*. 1974, vol. 19 of *Lecture Notes in Computer Science*, pp. 362–376, Springer.

[2] Gilles Kahn, "The semantics of a simple language for parallel programming," in *Information Processing*, 1974, vol. 74 of *Proc. IFIP Congress*, pp. 471–475.

[3] E.A. Lee and D.G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *Computers, IEEE Transactions on*, vol. C-36, no. 1, pp. 24 –35, Jan. 1987.

[4] G. Bilsen, M. Engels, R. Lauwereins, and J.A. Peperstraete, "Cyclo-static data flow," in *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, may 1995, vol. 5, pp. 3255 – 3258 vol.5.

[5] E.A. Lee and T.M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773 –801, May 1995.

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

[7] J.W. Janneck, I.D. Miller, D.B. Parlour, G. Roquier, M. Wipliez, and J.-F. Nezan, "Synthesizing hardware from dataflow programs," *Journal of Signal Processing Systems*, 2009.

[8] R. Gu, J. Janneck, M. Raulet, and S. Bhattacharyya, "Exploiting statically schedulable regions in dataflow programs," *Journal of Signal Processing Systems*, vol. 63, pp. 129–142, 2011.

[9] Matthieu Wipliez and Mickal Raulet, "Classification and transformation of dynamic dataflow programs," in *DASIP '10*, 2010.

[10] Matthieu Wipliez and Mickaël Raulet, "Classification of dataflow actors with satisfiability and abstract interpretation," *IJERTCS*, vol. 3, no. 1, pp. 49–69, 2012.

[11] J.L. Pino, S.S. Bhattacharyya, and E.A. Lee, "A hierarchical multiprocessor scheduling system for dsp applications," in *Twenty-Ninth Asilomar Conference on Signals, Systems and Computers, 1995.*, 1995, pp. 122–126.

[12] Jani Boutellier, Mickal Raulet, and Olli Silvn, "Automatic hierarchical discovery of quasi-static schedules of rvc-cal dataflow programs," *Journal of Signal Processing Systems*, pp. 1–6, 2012.

[13] Johan Ersfolk, Ghislain Roquier, Johan Lilius, and Marco Mattavelli, "Scheduling of dynamic dataflow programs based on state space analysis," in *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, march 2012, pp. 1661 – 1664.

[14] J.W. Janneck, "A machine model for dataflow actors and its applications," in *Signals, Systems and Computers (ASILOMAR), 2011 Conference Record of the Forty Fifth Asilomar Conference on*, nov. 2011, pp. 756 –760.

[15] Johan Ersfolk, Ghislain Roquier, Fareed Jokhio, Johan Lilius, and Marco Mattavelli, "Scheduling of dynamic dataflow programs with model checking," in *IEEE International Workshop on Signal Processing Systems (SiPS)*, 2011.