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

Quasi-static scheduling of CAL actor networks for reconfigurable video coding

Jani Boutellier · Christophe Lucarz · Sébastien Lafond · Victor Martin Gomez · Marco Mattavelli

Received: date / Accepted: date

Abstract The upcoming Reconfigurable Video Coding (RVC) standard from MPEG (ISO / IEC SC29WG11) defines a library of coding tools to specify existing or new compressed video formats and decoders. The coding tool library has been written in a dataflow/actor-oriented language named CAL. Each coding tool can be represented with an extended finite state machine and the dependencies between the tools are described as dataflow graphs. This paper proposes an approach to model the CAL actor network with Parameterized Synchronous Data Flow and to derive a quasi-static multiprocessor execution schedule for the system. In addition to proposing a scheduling approach for RVC, an extension to the well-known permutation flow shop scheduling problem that enables rapid run-time scheduling of RVC tasks, is introduced.

Keywords scheduling · parallel processing · digital signal processors · modeling

J. Boutellier
Machine Vision Group
University of Oulu, Finland
E-mail: jani.boutellier@ee.oulu.fi

C. Lucarz
Microelectronic Systems Laboratory
École Polytechnique Fédérale de Lausanne, Switzerland
E-mail: christophe.lucarz@epfl.ch

S. Lafond
Embedded Systems Laboratory
Åbo Akademi University, Finland
E-mail: sebastien.lafond@abo.fi

V. Martin
Machine Vision Group
University of Oulu, Finland
E-mail: victor.martin@ee.oulu.fi

M. Mattavelli
Microelectronic Systems Laboratory
École Polytechnique Fédérale de Lausanne, Switzerland
E-mail: marco.mattavelli@epfl.ch

1 Introduction

The effort of designing the Reconfigurable Video Coding (RVC) standard [1] is motivated by the intent to describe existing video coding standards with a set of common atomic building blocks (*e.g.*, IDCT). Under RVC, existing video coding standards are described as specific configurations of these atomic blocks. This greatly simplifies the task of designing future multi-standard video decoding applications and devices by allowing software and hardware reuse across video standards.

The RVC coding tools are specified in a dataflow/actor object-oriented language named CAL [2] that describes the atomic blocks in a modular way and exposes parallelism between computations. Abstract, high-level models require a systematic implementation methodology and tools to realize into practical systems. Design flows are generally composed of several phases: specification, design space exploration (DSE) and implementation.

The implementation phase needs two components: the implementation code (generally C and VHDL) and a schedule. Code generators are under development in the RVC framework [3,4]. A fundamental step to efficiently complete the implementation phase supported by the code generators is the schedule, *i.e.* the sequence in which CAL actors fire.

This paper introduces a methodology to derive quasi-static execution schedules for a set of CAL actor networks, including the already released RVC MPEG-4 Simple Profile video decoder. In quasi-static scheduling most of the scheduling effort is done off-line and only some infrequent data-dependent scheduling decisions are left to run-time. The off-line determined schedules are collected to a repository that is used by the run-time system, which selects entries from the repository and appends them to the ongoing program execution. This approach limits the number of run-time scheduling decisions and improves the efficiency of the system.

2 Related work

2.1 The Reconfigurable Video Coding framework

The RVC framework uses CAL models for defining video coding tools. The Open Dataflow environment [5] supports the design, the simulation and the debugging of CAL models. Deriving software and/or hardware implementations from these CAL models is a non trivial task. However, several tools already exist: a hardware code generator converts CAL actors in a mixed VHDL and Verilog implementation [3,6]. The results are very promising and can outperform manual designs in terms of area and throughput. A software code generator converts CAL actors into a C/C++ implementation [4]. The hardware and software code generators have the capability to compile networks of actors. Currently, the software code generator converts CAL actors to C language and inserts automatically a dynamic scheduler written in SystemC in order to determine at run-time the order of execution of the actors. However, determining the schedule at run-time is a time consuming task which may considerably deteriorate the system performance. Another tool (Syndex [7]) focuses on multiprocessor implementation. Syndex maps Synchronous Data Flow (SDF) graphs onto multiprocessor architectures. The problem is that CAL is an asynchronous model and cannot be used with this tool. A schedule of the actors is missing.

This paper helps in building an efficient implementation from CAL models. Because deciding the entire schedule at run-time introduces too much overhead, the idea is to reduce this overhead by scheduling some parts of the model off-line. Knowing the exact schedule of

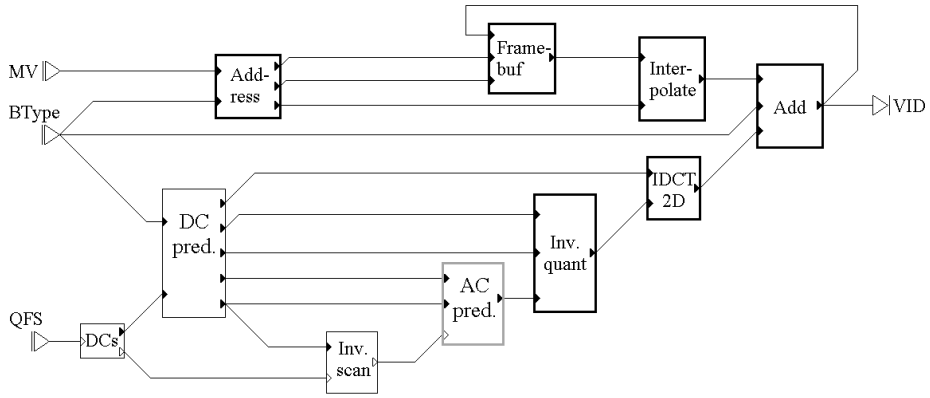

Fig. 1 High level view of the RVC MPEG-4 Simple Profile decoder in the RVC framework.

the model means also transforming the asynchronous model CAL into a SDF representation which can be used by the Syndex tool, enabling multiprocessor implementation from the high level model CAL. The work described in this publication is an updated and extended version of a previous work [8]. This publication introduces a more strictly defined model for scheduling CAL actor networks and expresses the resulting model with the notation that is used in Parameterized Synchronous Data Flow [9].

2.2 Scheduling of similar systems

CAL models are too general to be scheduled efficiently; by default, the scheduling of actor functions is resolved only at run-time. Each CAL actor is represented as an extended finite state machine (EFSM) that contains variables and guard conditions that enable or disable possible state transitions. Actors are connected to each other with *dataflow edges* that are attached to the actor *ports*. Actors communicate by firing tokens along the edges of the dataflow graph. The state transition and token(s) fired by each actor is a function of the current state and the tokens that are available at its input.

As a partial solution to the scheduling problem, literature suggests that EFSMs can be transformed into regular FSMs, with the cost of a possible state-space explosion [10]. For networks of FSMs it would then be possible to use modeling methods such as [11], [12] or [13].

Recently, also two other projects have been started to develop a methodology for deriving efficient schedules for CAL actor networks. The approach of von Platen and Eker [14] sketches a method to classify CAL actors to different dataflow classes for efficient scheduling. Gu *et al.* [15] introduce a way to extract statically schedulable regions from CAL networks. Both of these approaches, as well as the one presented in this publication, try to minimize the number of run-time scheduling decisions.

3 The proposed approach

Figure 1 shows a high-level view of the RVC implementation of the MPEG-4 Simple Profile decoder and Table 1 provides an overview of our approach. The first three steps of our

Table 1 Overview of our approach.

Step	Explanation
1	Classification of CAL actors: regular, borderline or dynamic
2	Unrolling of K actors: from K EFSMs to $K * L$ SDF graph fragments
3	SDF graph merging: merge $K * L$ SDF graph fragments to L system-level SDF graphs
4	Processor assignment: assign each SDF actor of the L system-level graphs to a processor
5	Off-line scheduling: compute static schedule for L system-level SDF graphs
6	Execution and run-time scheduling

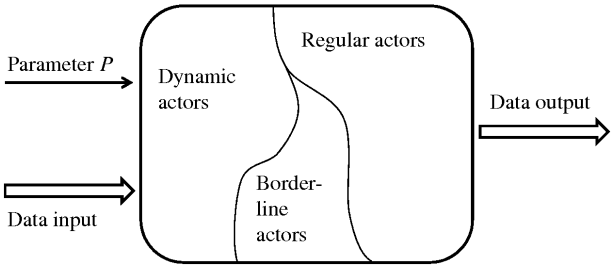

Fig. 2 The high-level model of the system to be scheduled.

approach describe a set of procedures that transform the CAL actor network into a set of statically schedulable SDF graphs. They are described in Sections 3.2, 3.3, and 3.4; the other steps, described in Section 5, comprise the off-line and on-line scheduling mechanism.

3.1 Preprocessing

The scheduling approach proposed in this publication requires some pre-processing of the CAL actor network to acquire information that is necessary for the graph transformations. First, the CAL actor network must be analyzed to extract information about the token rates of actor ports: for each port of all actors in the system, it must be known if the token production rate and consumption rate can be determined quasi-statically or not.

Determining the quasi-static behaviour requires exploring the state machine of the actor. The EFSM is assumed to have an *initial state*, from which several state transition paths depart. It is also assumed that all of the departing state transition paths eventually lead back to the initial state. The CAL actor is quasi-statically schedulable if the (1) state transitions, (2) token production and (3) consumption rate of each path are static and not affected by any data. An example of a state transition path can be seen in Figure 3: *newVop* $\Rightarrow$ *other* $\Rightarrow$ *other*.

Our scheduling approach has a strong assumption about the nature of the CAL network that is to be scheduled. This is illustrated in Figure 2 from a high level. The model has a dataflow input and output, as well as a parameter P. The parameter P is a special kind of a dataflow input that changes the configuration of the CAL actor network according to the value of P. P needs to be clearly identified and defined before applying our approach to the network. P can be combined of any fixed, non-negative number of dataflow inputs and every value that P may get, must be known at system design time.

Our scheduling algorithm creates a static schedule for every possible value of P. *E.g.*, if P is a single constant, only one schedule is created. On the other hand, if P can acquire a

great variety of values or is a combination of several parameters, the number of generated schedules can become considerable. In the MPEG-4 SP application example (Figure 1), *BType* works as the parameter P. Based on the CAL code the *BType* token can get 4096 different values, but inspection of the whole decoder network reveals that only 4 separate values affect the scheduling of the network. The number of different values of P is denoted with the constant L.

For this work the port token rates and the specification of P were determined manually. Automating the acquisition of this information is left outside the scope of this work.

3.2 Classification of actors

As a first step of our approach, we classify each actor in the CAL actor network to one of three categories: regular actors, borderline actors and dynamic actors. The classification is done solely based on the *a-priori* information of the behaviour of the ports of each actor. We assume that each actor has been analyzed beforehand so that each port is known to have a quasi-static or variable token rate.

If an actor has a quasi-static token rate on every input and output port, it is classified as a *regular* actor. If at least one of the actor *input* ports has a variable token rate, it is classified as a *borderline* actor. Actors that do not satisfy these conditions are classified as *dynamic* actors. In Table 1, the number of regular *and* borderline actors is denoted by K.

Regular actors require some explanation: a regular actor can have several operation modes that are switched by the parameter P by selecting the state transition path from the initial state. However, the token rate of each port may only vary as a function of P. As an example, we can take the *add* actor of MPEG-4 SP. When $P = V_1$, the actor consumes 64 tokens from input port [TEX] and produces 64 tokens through the output port [VID]. When $P = V_2$, the actor consumes 64 tokens from port [MOT] and produces 64 through [VID], but does not consume any tokens from [TEX].

At the actor network level, a strict rule is imposed regarding the actor classes: borderline actors may only have dynamic actors connected to their input ports. Similarly, regular actors may only have borderline or dynamic actors connected to their input ports. Regular actors and borderline actors may only be connected to regular actors through their output ports.

This requirement is illustrated in Figure 2. Actors that do not satisfy this requirement, must be re-classified so that the requirement is met before proceeding to next steps in our approach. An example of re-classification: if actor A has been initially classified as a regular actor, but the output ports of A are connected to a dynamic actor B, it is mandatory to re-classify A as a dynamic actor before proceeding.

Figure 1 depicts the result of our actor classification for the RVC implementation of the MPEG-4 Simple Profile decoder. Actors with thick outlines are regular actors and the ones with thin outlines are dynamic actors. Actor ports colored with black triangles are ports with quasi-static data rates and ports with white triangles are ones with variable data rates. Because of a variable rate input port, the *AC prediction* actor has been classified as a borderline actor. The *DC prediction* actor is initially classified as a regular actor, but because one of its outputs is connected to a dynamic actor, *DC prediction* has been re-classified as dynamic.

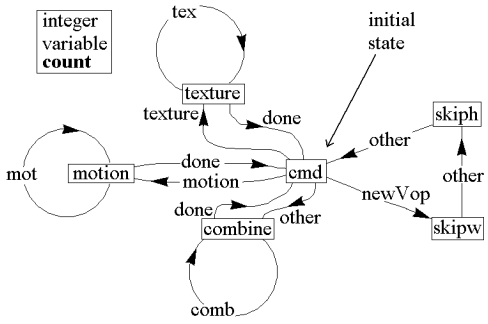

Fig. 3 The EFSM of the *add* actor.

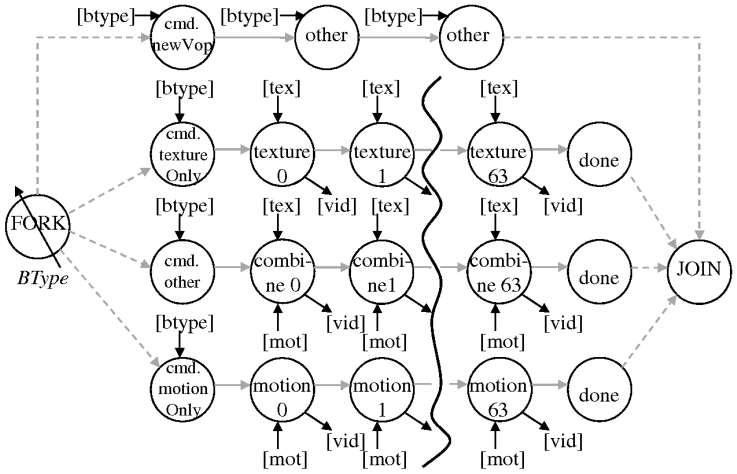

Fig. 4 The four SDF graph fragments acquired from unrolling *add*.

3.3 EFSM unrolling

In Step 2 of our approach, the EFSM representations of *regular* CAL actors are unrolled into a collection of SDF graphs, so that every state transition becomes an SDF actor. EFSMs are assumed to have an *initial state* that serves as the origin of unrolling. It is assumed that all state transition paths leaving from the initial state eventually return back to the initial state. The number of state transitions (l) originating from the initial state is assumed to be $1 \leq l \leq L$. As an example, in Figure 3, $l = L = 4$.

We call the SDF graph that is produced from unrolling one state transition path an SDF *graph fragment*. There is no possibility to represent dynamic control structures in SDF, but for intuitiveness the graph fragments can be illustrated as being connected by a fork-actor. The unrolled version of the Figure 3 EFSM is shown in Figure 4.

If a CAL actor does not have a dataflow input for P, the actor is still required to produce L graph fragments. In this case the L graph fragments will be identical. When $1 < l < L$, only some of the graph fragments will be identical to each other.

The SDF actors within one graph fragment are interconnected by *control flow edges*. Control flow edges do not transmit data, they just make sure that the dependencies expressed in CAL actors are also observed in scheduling. Nevertheless, when the SDF graph is later

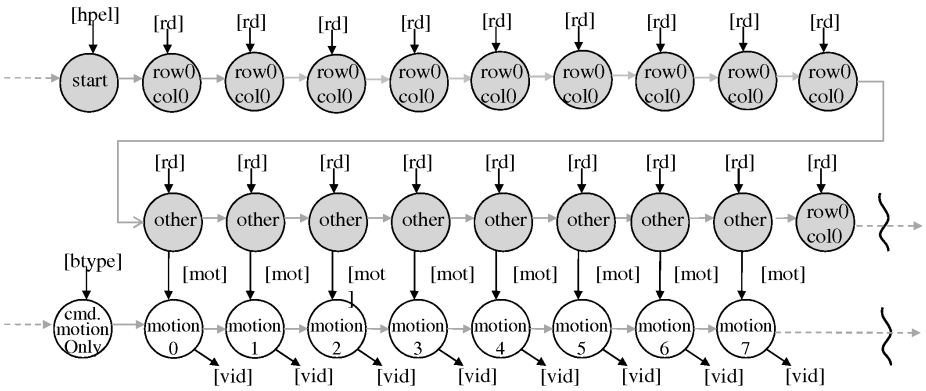

Fig. 5 Joining the graph fragments of *interpolate* and *add* for one value of *P*.

scheduled, control edges behave in the same manner as other edges. In the figures of this publication, control edges have a gray color.

Dynamic actors are omitted during the unrolling phase and thus they do not contribute to the creation of SDF graph fragments. Borderline actors are handled in a special fashion: a set of *stub* SDF actors are produced from borderline CAL actors. For each output port of a borderline actor there will be a set of stub actors. The number of SDF actors in a stub actor set is equal to the token rate of the port. The stub actors provide a link between the statically and the dynamically scheduled code regions.

The state transition paths of EFSMs may contain iterative self loops such as *comb* in Figure 3. This poses no problem if the number of loop iterations is fixed. Variable numbers of iterations can not be supported by the proposed approach. Another issue arises when the number of state transitions is fixed and large, which respectively results in a large SDF graph. However, the large SDF graphs only exist during off-line scheduling: in the run-time system the program code represented by the SDF actor does not need to be replicated but can be replaced by repetitive calls to the same function.

All of the actors in the MPEG-4 Simple Profile decoder do not map trivially into our model. One exception can be found within the hierarchical *IDCT2D* actor. One of the actors contained by *IDCT2D* is named *Clip* and is responsible for saturating integer values. The problem with the implementation of this actor is that depending on the value of the integer that has to be clipped, a different state transition is chosen. This is a feature that is not supported by our scheduling approach. In the work of Gu *et al.* [15] the same problem has been noticed and corrected by using a modified, predictable version of the same actor. We have chosen the same solution to this particular problem.

3.4 Parameter-specific system-level graphs

The unrolled EFSM representations of the CAL actors consist of SDF graph fragments that can be thought to be connected by a *fork* actor (see Figure 4). The SDF actors in Figure 4 have the same data flow interfaces as the state transitions in the CAL code. In the third step of our approach these data flow interfaces are connected to the respective interfaces of graph fragments originating from other CAL actors to create *system-level* SDF graphs. An

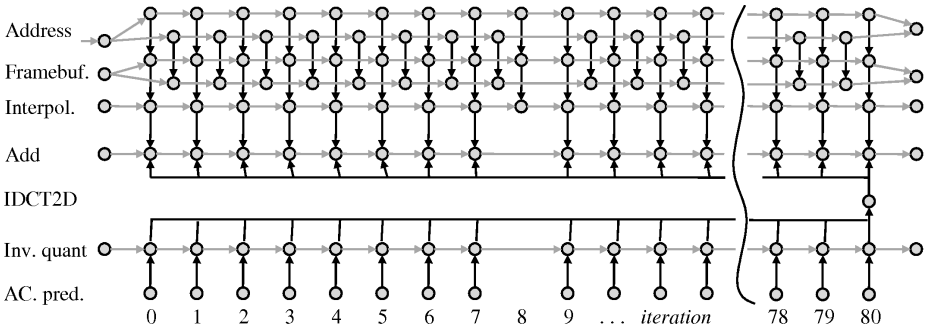

Fig. 6 A system-level graph corresponding to one value of P.

example of joining graph fragments of two CAL actors is depicted in Figure 5. The gray vertices belong to the *interpolate* actor and the white vertices belong to the *add* actor.

Since there is one SDF graph fragment in every unrolled CAL actor for each value of P, the SDF graph joining happens between fragments that represent the same value of P. *E.g.*, the SDF graph fragment of *add* that represents P value l is joined with the SDF graph fragment of the *interpolation* actor that represents P value l. Thus, for each value of parameter P, there will be a unique system-level graph. Figure 6 shows a system-wide subgraph that contains all the actions and their dependencies.

4 PSDF modeling

Before discussing the scheduling of the system-level graphs, we briefly describe the Parameterized Synchronous Data Flow [9] (PSDF) computation model that is used to describe our scheduling problem. PSDF extends the SDF computation model by allowing run-time reconfiguration of SDF graphs by parameter changes. In PSDF terminology the parameterized SDF graph that models the functional behaviour of the system, is called the *body graph*. The parameters of the body graph are configured by two other graphs: the *init graph* and the *subinit graph*. The set of these three graphs defines a PSDF *subsystem*.

The subinit graph can change those parameters of the body graph that are not connected to the outside interface of the subsystem and it is executed once before each invocation of the body graph. The init graph is more powerful as it can also change the interface parameters of the PSDF subsystem. However, it is only executed once before each execution of the subsystem's *parent* graph.

In this work we use the parametrization only on token rates of edges. By parameterizing some edges it is possible to enable and disable certain paths of execution and implement a sort of if-then-else functionality. Guidelines of doing this have previously been shown in the work of Bhattacharya and Bhattacharyya [16] on page 133.

4.1 PSDF model of the system

Figure 7 shows a PSDF model of the MPEG-4 SP decoder that consists of a subinit graph, init graph and a body graph. The body graph contains 6 actors, of which 4 are hierarchical and thus marked with a double rectangle. These actors are named S_1, S_2, S_3 and S_4; we

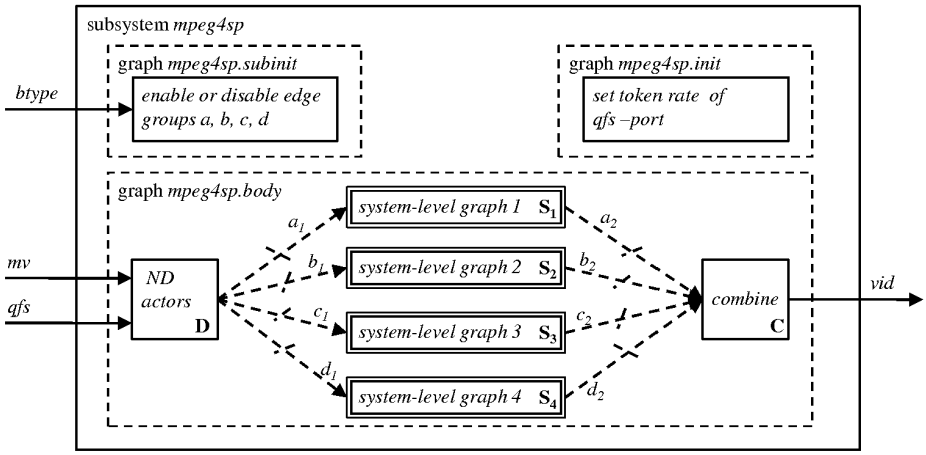

Fig. 7 The PSDF model of the RVC MPEG-4 SP decoder system.

shall denote the set of these four actors by S. The four actors of S contain the system-level SDF graphs that were assembled as described in Subsection 3.4. We shall call the set of SDF actors inside S with the name T. The non-hierarchical actor labeled D contains all the functionality that was classified in Subsection 3.2 as borderline (except for *stubs* that are a part of S) or dynamic actors. Since the execution time of the D is unpredictable, it is not modeled in higher detail here.

The actors S are connected to D by dashed edges. The dashed edges denote that the existence of these edges depends on the parameter values a, b, c and d that are toggled in the subinit graph. One way to realize the parameterized existence of these edges, is to change the token production rate of the respective output port of D to zero when the edge is not needed. The parameterized edges a, b, c and d have been marked with a slash to denote that the edges actually represent a set of edges. The parameterized edges represent the same functionality on the system-level as the *fork* actor (see Figure 4) did on actor-level.

The init graph changes the interface token rate of the port *qfs*, because it may change on every invocation of the subsystem's parent graph. Finally, the actor labeled C serves as a common output that gathers the data produced by the four optional actors S.

5 Scheduling

In this section we show a way to create efficient (multiprocessor) schedules for systems such as the one shown in Figure 7. The scheduling consists of two parts: the off-line (design time) part and the run-time part.

We assume that the run-time system consists of a set of homogeneous or heterogeneous processors that is fixed at compile time (*i.e.* we do not account for the possibility of a processor failing or the addition of extra processors during run-time).

5.1 Off-line scheduling

The first step of off-line scheduling is to assign each actor of T to one of the processors in the system. Each actor is mapped to exactly one processor; each processor may be respon-

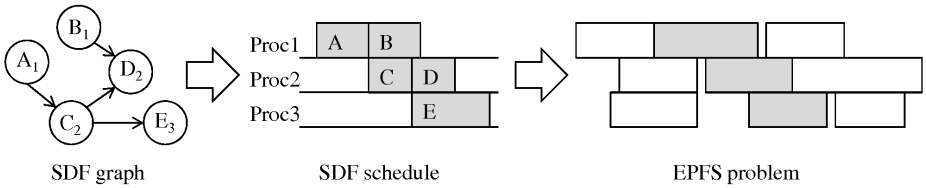

Fig. 8 Run-time pipelining of static SDF graph schedules.

sible for any number of actors. A majority of the actors of T originate from unrolled loops. This means that there are several instances of the same function. Evidently, it is advisable to map these instances of the same functionality to the same processor, even if it is not mandatory. The actor instances of the same function are connected with control flow edges (See Subsection 3.3), which ensures a correct order of execution.

To produce fully defined off-line schedules, the latency of each actor in T must be fixed. If the process modeled as an actor has some variance in the latency, the SDF actor must assume the worst-case latency for successful off-line scheduling. Fixed execution times are especially beneficial for generating multiprocessor schedules, since it allows inter-processor communication issues to be resolved off-line.

The scheduling of the functionality inside D is not considered here since it requires a completely different scheduling approach, which is out of the scope of this publication. Here, the functionality of D can be assumed to be executed on a single processor in a sequential fashion, which does not require any special methods. In contrast, the graphs S are scheduled off-line using a fully static scheduling approach. We do not discuss any of these here, since there are plenty of off-line scheduling methods available. The book of Sriram & Bhattacharyya [17] contains a good overview of available methods.

5.2 On-line scheduling

On-line scheduling takes place when the system is actually running and computing. In the approach presented in this paper, the run-time scheduling effort is actually limited to selecting a pre-computed schedule out of the ones generated and stored at system design time.

With regard to Figure 7, the suitable schedule is selected based on the token that comes from the dataflow input to the subinit graph. According to the theoretical view of the system, the schedule is selected by toggling the parameterized edges (a, b, c and d) and enabling *one* of the graphs in S.

At run-time the system consecutively executes pre-computed schedules in an order which is unknown at system design time. This raises a question how to merge two consecutive multiprocessor schedules together at run-time. Figure 8 illustrates the problem as a sequence of steps: the leftmost step shows an arbitrary SDF graph, where the actors have already been assigned to processors (according to the subscript). In the next step a static schedule has been derived for the SDF graph. The last step shows the outline of this static schedule between two other arbitrary schedules.

The situation depicted in the last step is related to *pipelining*, which allows keeping the *utilization* of the processors high. The PSDF model itself does not offer a way to pipeline the static schedules on multiprocessor systems. Fortunately, the problem resembles closely the Permutation Flow Shop (PFS) scheduling problem [18], which has previously been applied

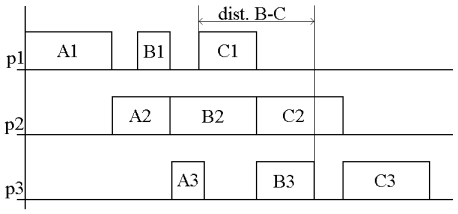

Fig. 9 A conventional PFS schedule.

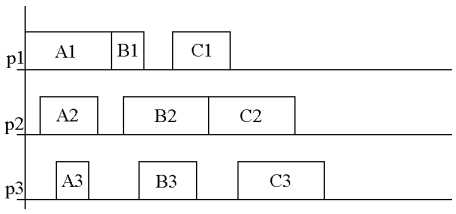

Fig. 10 An EPFS schedule.

to signal processing applications [19]. However, the PFS model is not directly applicable to the problem at hand. This issue will be explained and solved in the next subsection.

5.3 Extended permutation flow shop scheduling

Flow shop scheduling is a specific type of multiprocessor scheduling that has very elegant theoretical properties that make it practical for problems like the one shown here. We are given N jobs to be scheduled on M processors. Each job consists of M tasks, and the jth task in the job must be scheduled on processor j. A job can issue its jth task to processor j if the $(j-1)$st task is complete and processor j is free. Each task is assumed to have a predetermined constant processing time. By definition, each job must have M tasks, one for each processor. However, by setting the execution time of a task to zero, the effect is the same as if that task would not exist. This is called *machine skipping* in literature.

Permutation flow shop scheduling is a more restricted version of flow shop scheduling. Here, task j must be performed for job $n-1$, before it can be performed for job n. Usually the goal in solving the PFS problem is to find a permutation of jobs which minimizes the makespan (total schedule length in time units). We also assume that this is the objective.

Figure 9 shows the Gantt-chart of a PFS problem instance consisting of 3 jobs, 3 processors, and 3 tasks per job. Each task within a job executes on a separate processor and no-wait timetabling [18] ensures that the next task within the same job starts immediately upon completion of the previous task in the same job. The inter-job distance is the overlap in time between two sequential jobs, and is shown for jobs B and C in Figure 9.

A PFS job resembles closely a multiprocessor schedule of an SDF graph (see Figure 8), except for the fact that in SDF schedules several processors can work on the same job simultaneously, which is not possible in PFS. PFS gives a solution to the problem of merging consecutive, static multiprocessor schedules together. We are now going to introduce a small modification to the PFS assumptions to allow its use for modeling multiprocessor SDF graph schedules. We shall call this modified PFS model the *extended permutation flow shop* (EPFS) model. EPFS extends PFS in two respects and is capable of representing a larger set of scheduling problems.

Previous research [19] suggests that a particularly efficient implementation of run-time no-wait PFS scheduling can be made possible by pre-computing the inter-job distances at compile-time and storing them into a storage I. The first extension to PFS is to *enable* dependencies between tasks by explicitly determining the distances in I. For example, the inter-job distance could be increased so that $C1$ is forced to start only after $B2$ finishes (see Figure 9). No-wait PFS cannot represent these types of dependencies. This extension is only possible when all job types are known at system design time.

The second extension *disables* dependencies between tasks. In addition to the inter-job distance storage I this requires another storage J. We assume that the storage J explicitly defines the distance between tasks within one job. In the no-wait timetabling definition this distance is fixed to zero. The EPFS model breaks this limitation by allowing non-zero offsets between tasks within a job. This is not allowed in the PFS model, since it assumes that the next task starts *after* the previous task has finished.

With the EPFS model it is possible to model static multiprocessor SDF schedules and efficiently pipeline the schedules at run-time.

6 Experiments

The transformation and off-line scheduling steps described in Sections 3 and 5 have been implemented to a great extent in Java. In the earlier phases of the transformations, the EFSMs are represented with classes provided by the JGraphT package [20] and during the later stages the SDF graphs are represented with classes from the SDF4J package [21]. All of these steps are performed in the same OpenDF environment as the code generators [4, 3] use, which enables smooth interoperability. However, the practical work to enable the use of these schedules for code generation has not yet been started. The on-line scheduling method has been implemented in the C language and there is also a yet unpublished hardware implementation of it.

7 Discussion

The preprocessing steps discussed in Subsection 3.1 are done completely manually at the moment. However, some work has been done to derive the token communication patterns automatically, but this functionality has not yet been tested in conjunction with our scheduling approach.

The strict requirements of the assumed system (See Subsection 3.2) allow efficient scheduling for systems that have dynamic functionality only before the regular functionality. Generally, there can be CAL actor networks that might have a dynamic part in the end of the network or consist of several mixed regular and dynamic patches. Extending our approach to such general systems is a clear direction for future work.

8 Conclusion

This paper describes a sequence of steps that allow deriving quasi-static schedules for a set of CAL actor networks, such as the MPEG-4 Simple Profile decoder in Reconfigurable Video Coding. The procedure is based on local and global graph transformations followed by off-line and on-line multiprocessor scheduling. At run-time, the piecewise static schedules are selected based on the system parameters, and appended to the ongoing processor schedule by means of extended flow shop scheduling.

Acknowledgements This research has been partially funded by the Nokia Foundation.

References

1. C. Lucarz, M. Mattavelli, J. Thomas-Kerr, and J. W. Janneck, "Reconfigurable media coding: a new specification model for multimedia coders," in *IEEE Workshop on Signal Processing Systems*, Shanghai, China, October 2007, pp. 481–486.
2. J. Eker and J. W. Janneck, "Cal language report, tech. memo ucb/erl m03/48," 2003.
3. J. W. Janneck, I. Miller, D. Parlour, M. Mattavelli, C. Lucarz, M. Wipliez, M. Raulet, and G. Roquier, "Translating dataflow programs to efficient hardware: an mpeg-4 simple profile decoder case study," in *DATE Workshop of The New Wave of the High Level Synthesis*, Munich, Germany, March 2008.
4. M. Wipliez, G. Roquier, M. Raulet, J.-F. Nezan, and O. Deforges, "Code generation for the mpeg reconfigurable video coding framework: From cal actions to c functions," in *IEEE International Conference on Multimedia and Expo*, Hannover, Germany, April 2008, pp. 1049–1052.
5. "Open dataflow sourceforge project, http://opendf.sourceforge.net/."
6. C. Lucarz, M. Mattavelli, M. Wipliez, G. Roquier, M. Raulet, J. W. Janneck, I. D. Miller, and D. B. Parlour, "Dataflow/actor-oriented language for the design of complex signal processing systems," in *Conference on Design and Architectures for Signal and Image Processing*, Bruxelles, Belgium, November 2008, pp. 168–175.
7. M. Raulet, M. Babel, O. Deforges, J. Nezan, and Y. Sorel, "Automatic coarse-grain partitioning and automatic code generation for heterogeneous architectures," in *IEEE Workshop on Signal Processing Systems*, Seoul, Korea, August 2003, pp. 316–321.
8. J. Boutellier, V. Sadhanala, C. Lucarz, P. Brisk, and M. Mattavelli, "Scheduling of dataflow models within the reconfigurable video coding framework," in *IEEE Workshop on Signal Processing Systems*, Washington D.C, USA, October 2008, pp. 182–187.
9. B. Bhattacharya and S. Bhattacharyya, "Parameterized dataflow modeling for dsp systems," *IEEE Transactions on Signal Processing*, vol. 49, no. 10, pp. 2408–2421, Oct 2001.
10. O. Henniger and P. Neumann, "Test case generation based on formal specifications in estelle," in *IEEE International Workshop on Factory Communication Systems*, Leysin, Switzerland, October 1995, pp. 135–141.
11. A. Girault, B. Lee, and E. A. Lee, "Hierarchical finite state machines with multiple concurrency models," *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, vol. 18, pp. 742–760, 1999.
12. L. Thiele, K. Strehl, D. Ziegenbein, R. Ernst, and J. Teich, "Funstate—an internal design representation for codesign," in *IEEE/ACM international conference on Computer-aided design*, San Jose, California, USA, November 1999, pp. 558–565.
13. J. Buck and R. Vaidyanathan, "Heterogeneous modeling and simulation of embedded systems in el greco," in *Eighth international workshop on Hardware/software codesign*, San Diego, California, USA, May 2000, pp. 142–146.
14. C. von Platen and J. Eker, "Efficient realization of a cal video decoder on a mobile terminal," in *IEEE Workshop on Signal Processing Systems*, Washington D.C, USA, October 2008, pp. 176–181.
15. R. Gu, J. W. Janneck, M. Raulet, and S. S. Bhattacharyya, "Exploiting statically schedulable regions in dataflow programs," in *IEEE International Conference on Acoustics, Speech and Signal Processing*, Taipei, Taiwan, April 2009, to appear.
16. B. Bhattacharya and S. S. Bhattacharyya, "Parameterized modeling and scheduling of dataflow graphs," Institute for Advanced Computer Studies, University of Maryland at College Park, Tech. Rep. UMIACS-TR-99-73, December 1999, also Computer Science Technical Report CS-TR-4083.
17. S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*. New York, USA: Marcel Dekker, 2000.
18. S. French, *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop*. Chichester, UK: Ellis Horwood Limited, 1982.
19. J. Boutellier, S. S. Bhattacharyya, and O. Silven, "Low-overhead run-time scheduling for fine-grained acceleration of signal processing systems," in *IEEE Workshop on Signal Processing Systems*, Shanghai, China, October 2007, pp. 457–462.
20. "Jgraph sourceforge project, http://sourceforge.net/projects/jgraph."
21. "Sdf4j dataflow sourceforge project, http://sourceforge.net/projects/sdf4j."