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 (actor) can be represented with an extended finite state machine and the data communication 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 Gomez
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] was 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.

The CAL model of computation is untimed [3], which means that the order of events in a CAL specification reflects ordering induced by causality; there is no particular schedule. In practice, this means that various control mechanisms [4] need to ensure at run-time that events take place in a correct order. Such run-time decision making makes the specification flexible, but creates overhead and unpredictability.

As a remedy, we introduce a methodology to derive quasi-static execution schedules for a set of CAL actor networks, including the 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

Similar models of computation, as the networked CAL actors, have been proposed in [5], [6] and [7]. These models of computation integrate finite state machines (FSM) with dataflow graphs [8]. CAL actors can be expressed as FSMs that also contain variables. This makes the CAL FSMs extended finite state machines that can be transformed into regular FSMs, with the cost of a possible state-space explosion [9].

The CAL EFSMs contain run-time control mechanisms [4] that allow or deny 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 idea of this work is to reduce the run-time overhead of the targeted CAL networks by analyzing the network behaviour off-line and thus replace the run-time state transition control mechanisms by a set of static schedules. Our approach relies on the assumption that the CAL EFSMs have a limited state-space, which is true for the MPEG-4 SP decoder model. The static scheduling is made possible by transforming the untimed CAL model into a set of homogeneous synchronous dataflow (HSDF) [10] graphs that are statically scheduled at design time. The schedules are stored and invoked at run-time on demand, which allows some data-dependence in the run-time control flow of the CAL networks. In literature this is called quasi-static scheduling.

The work described in this publication is an updated and extended version of a previous work [11]. 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 [8]. Recently, also two other approaches have been proposed
to develop a methodology for deriving efficient schedules for CAL actor networks. The approach of von Platen and Eker [12] sketches a method to classify CAL actors to different dataflow classes for efficient scheduling. Gu et al. [13] introduce a way to extract statically schedulable regions from CAL networks. Both of these approaches, as well as the one presented in this work, try to minimize the number of run-time scheduling decisions.

The Open Dataflow environment [14] 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 to a hardware description [15] and a software code generator converts CAL actors into their C/C++ implementation [16]. The hardware and software code generators have the capability to compile networks of actors. Currently, the software code generator converts CAL actors to the 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.

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 approach describe a set of procedures that transform the CAL actor network into a set of statically schedulable HSDF graphs. They are described in Sections 3.2, 3.3, and 3.4. Sec-

<table>
<thead>
<tr>
<th>Step</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Classification of CAL actors: regular or dynamic</td>
</tr>
<tr>
<td>2</td>
<td>Unrolling of K actors: from K EFMSs to K*</td>
</tr>
<tr>
<td>3</td>
<td>HSDF graph merging: merge K*</td>
</tr>
<tr>
<td>4</td>
<td>Processor assignment: assign each HSDF actor to a processing element</td>
</tr>
<tr>
<td>5</td>
<td>Off-line scheduling: compute a static schedule for</td>
</tr>
<tr>
<td>6</td>
<td>Execution and run-time scheduling</td>
</tr>
</tbody>
</table>

Table 1 Overview of our approach.
tion 4 describes the PSDF model of the MPEG-4 SP decoder. Section 5 describes the off-line and on-line scheduling of the decoder model.

3.1 Preprocessing

The scheduling approach proposed in this work requires some preprocessing of the CAL actor network to acquire information that is necessary for the graph transformations. Figure 2 shows the abstract model of the assumed CAL network. 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. These strict requirements imposed on $P$ are required by our techniques and are not a part of the PSDF model that allows more flexible parametrization. The definition of $P$ is done manually at the moment.

For an arbitrary actor network, the parameter $P$ can be identified as follows: every edge contributing to $P$ must come from outside to the network that is to be scheduled. All changes in the network topology or actor token rates must happen as a function of $P$. On the other hand, $P$ must not contain edges that do not cause network topology or token rate changes.

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 five different network topologies are produced; many values produce an identical actor network. Evidently, the $BType$ values that result in identical networks, should be mapped to a single $P$ value. The number of different values of $P$ is denoted with the notation $|P|$.

On the actor level, the network must be analyzed to extract information about the token rates of actor ports for each mode of $P$. For each port of all actors in the system, the $P$-value dependent token production and consumption rate must be known, or be specified as variable. Determining the token rates requires exploring the state machines of the actors. The EFSMs are 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. An example of a state transition path can be seen in Figure 4: $newVop \Rightarrow other \Rightarrow other$. 

![Fig. 2 The abstract model of the system to be scheduled.](image-url)
3.2 Actor classification

If the token consumption and production rate for all state transition paths of an actor are constant for all values of $P$, the CAL actor is classified as regular. If any of the actor ports has a non-constant token rate with any value of $P$, the actor is dynamic. In Table 1, the number of regular actors is denoted by $K$.

Regular actors require some explanation: a regular actor may 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]. So, the token rates of ports for this actor do change, but only as a function of $P$.

As another example, Figure 3 depicts the EFSM of the inverse scan actor that is classified as dynamic. The EFSM state full has two departing state transitions (and it is not the initial state), which makes the token rates of several state transitions unpredictable. Thus, the inverse scan actor is classified as dynamic.

Furthermore, at the actor network level, our model assumes that the scheduled part of the CAL network matches the model in Figure 2. Verbally, the model is as follows: regular actors are not allowed to have dynamic actors connected to their output ports. If a CAL network does not match this network level requirement, there are two possibilities: 1) adjust the coverage of the scheduled network part or 2) reclassify individual actors so that the model requirements are met. An example of actor 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 $B$, $A$ can be re-classified as a dynamic actor to meet the requirements. This re-classification overrides the port token rates. In fact, both these two options have the same effect: actors outside the scheduled network, as well as dynamic actors, are left out of the further steps described in this work.

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. The DC prediction actor is initially classified as a regular actor, but because one of its outputs is connected to the dynamic Inverse scan actor, DC prediction has been re-classified as dynamic. The same applies to DC split.

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.
3.3 EFSM unrolling

Step 2 of our approach is automatic: the EFSM representations of regular CAL actors are unrolled into a collection of HSDF graphs, so that every state transition becomes an HSDF 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 $1 \leq l \leq |P|$. As an example, in Figure 4, $l = 4$.

We call the HSDF graph that is produced from unrolling one state transition path an HSDF graph fragment. There is no possibility to represent dynamic control structures in HSDF, but for intuitiveness the graph fragments can be illustrated as being connected by a fork-actor. The unrolled version of the Figure 4 EFSM is shown in Figure 5.
Fig. 6 Joining the graph fragments of *interpolate and add* for one value of \( P \).

If a CAL actor does not have a dataflow input for \( P \), the actor is still required to produce \( |P| \) graph fragments. In this case the \( |P| \) graph fragments will be identical. When \( 1 < l < |P| \), only some of the graph fragments will be identical to each other. In Figure 5, the last two graph fragments are identical. In reality these represent the block decoding modes with a non-zero motion vector and a zero motion vector.

The HSDF actors within one graph fragment are interconnected by static 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 HSDF graph is later scheduled, control edges behave in the same manner as dataflow edges. In the figures of this work, control edges have a gray color. Dynamic actors are omitted during the unrolling phase and thus they do not contribute to the creation of HSDF graph fragments.

The state transition paths of EFSMs may contain iterative self loops such as *comb* in Figure 4. This poses no problem if the number of loop iterations is fixed. Variable numbers of iterations within one \( P \) value 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 HSDF graph. However, the large HSDF graphs only exist during off-line scheduling: in the run-time system the program code represented by the HSDF actor does not need to be replicated but can be replaced by repetitive calls to the same function.

All of the CAL actors in the MPEG-4 Simple Profile decoder can not be directly scheduled by our approach. Such a case 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 [13] 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. The feedback loop that is seen in the CAL network of Figure 1, has fixed and known number of iterations and thus poses no problem in graph unrolling.
3.4 Parameter-specific system-level graphs

The unrolled EFSM representations of the CAL actors consist of HSDF graph fragments that can be thought to be connected by a *fork* actor (see Figure 5). The HSDF actors in Figure 5 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 automatically connected to the respective interfaces of graph fragments originating from other CAL actors, to create *system-level* HSDF graphs. An example of joining graph fragments of two CAL actors is depicted in Figure 6. The gray vertices belong to the *interpolate* actor and the white vertices belong to the *add* actor.

Since there is one HSDF graph fragment in every unrolled CAL actor for each value of \( P \), the HSDF graph joining happens between fragments that represent the same value of \( P \). For example, the HSDF graph fragment of *add* that represents \( P \) value \( i \) is joined with the HSDF graph fragment of the *interpolation* actor that represents \( P \) value \( i \). Thus, for each value of parameter \( P \), there will be a unique system-level graph. Figure 7 shows a system-level graph that contains all the actions and their dependencies. Table 2 shows the number of SDF actors (CAL actions) for each of the five system level graphs.

4 PSDF modeling

Before discussing the scheduling of the system-level graphs, we briefly describe the Parameterized Synchronous Data Flow [8] (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*.

### Table 2 Number of SDF actors in the five system level graphs.

<table>
<thead>
<tr>
<th>New frame</th>
<th>Inter block</th>
<th>ZeroMV block</th>
<th>Intra block</th>
<th>Inter and intra block</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>444</td>
<td>442</td>
<td>345</td>
<td>592</td>
</tr>
</tbody>
</table>
Fig. 8 The PSDF model of the RVC MPEG-4 SP decoder system.

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 [17].

4.1 PSDF model of the system

Figure 8 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 seven actors, of which five are hierarchical and thus marked with a double rectangle. These actors are named $S_1$, $S_2$, $S_3$, $S_4$ and $S_5$; we shall denote the set of these five actors by $S$. The five actors of $S$ contain the system-level HSDF graphs that were assembled as described in Subsection 3.4. We shall call the set of HSDF 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 dynamic actors. Since the execution time of $D$ 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 $a$, $b$, $c$, $d$ and $e$ that are toggled in the subinit graph. The parameterized existence of these edges is realized by changing the token production rate of the respective output port of $D$ to zero when the edge is not needed. The parameterized edges $a$, $b$, $c$, $d$ and $e$ 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 5) did on actor-level. The init graph could change the interface token rate of the port $qfs$, although it is fixed in MPEG-4
Fig. 9 Run-time sequencing of static SDF graph schedules. A, B, C, D and E are SDF actors or actor groups.

SP. Finally, the actor labeled C serves as a common output that gathers the data produced by the five optional actors S.

The work [18] presents a very similar modeling approach as the one described in this section. Their work is also based on PSDF semantics and the authors propose as well that the schedules of static graph partitions are computed at design time and are stored for use at run-time.

5 Scheduling

In this section we show a way to create efficient (multiprocessor) schedules for systems such as the one shown in Figure 8. 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 processing elements (PEs) that is fixed at design time (i.e. we do not account for the possibility of a PE failing or the addition of extra PEs during run-time). Scheduling encompasses the regular CAL actors within the CAL network that is to be scheduled. If the CAL network that is to be scheduled, is a part of a larger network, the outlying actors are not considered here.

5.1 Off-line scheduling

The first step of off-line scheduling is to assign each HSDF actor of T to one of the PEs in the system. Each actor is mapped to exactly one PE; each PE may be responsible for any number of actors. A majority of the actors of T originate from unrolled loops. This means that there are several calls to the same function. Evidently, it is advisable to map these instances of the same functionality to the same PE, 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 state transition of a CAL actor has some variance in the latency, the respective HSDF actor must assume the worst-case latency for successful off-line scheduling. Fixed execution times are especially beneficial for generating multi-PE schedules, since it allows inter-PE communication issues to be resolved off-line.

The scheduling of the functionality inside D is not considered here since it requires a fully dynamic run-time scheduling approach, which is out of the scope of this work. Here, the functionality of D can be assumed to be executed on a single PE in a sequential fashion, which does not require any special methods. In contrast, the graphs S are scheduled automatically off-line using a fully static scheduling algorithm. We do not discuss any of these here, since there are plenty of off-line scheduling methods available. The book of Sriram and Bhattacharyya [19] 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 work, 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 8, the suitable schedule is selected based on the token that comes from the dataflow input to the subunit graph. According to the view of the PSDF model, the schedule is selected by toggling the parameterized edges \((a, b, c, d \text{ and } e)\) 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 multi-PE schedules together at run-time. Figure 9 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 PEs 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 [20], which has previously been applied to signal processing applications [21]. 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 \(j\)th task in the job must be scheduled on processor \(j\). A job can issue its \(j\)th 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 (PFS) 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 this objective.
Figure 10 shows the Gantt-chart of a PFS problem instance consisting of three jobs (A, B, C), three processors (p1, p2, p3), and three tasks per job (B1, B2, B3 for job B). Each task within a job executes on a separate processor and no-wait timetabling [20] 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 10.

A PFS job resembles closely a multiprocessor schedule of an SDF graph (see Figure 9), except for the fact that in SDF schedules several processors can work on the same job simultaneously, which is not possible in PFS. 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 [21] 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 10). 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 (see Figure 11). This is not allowed in the PFS model, since it assumes that the next task starts *after* the previous task has finished.

When each system-level graph is defined as one EPFS job, it is possible to efficiently sequence the statically scheduled system-level graphs at run-time. Note that each EPFS job consists of a set of tasks that have been assigned to different PEs in the system (not every PE necessarily has a task). The actual functionality of each task depends on the HSDF actor \( \Rightarrow \) PE mapping that was set in Subsection 5.1.

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 [22] and during the later stages the HSDF graphs are represented with classes from the SDF4J package [23]. All of these steps are performed in the same OpenDF environment as the code generators [16, 15] use, which enables smooth interoperability.

The CAL simulator in the OpenDF environment has been modified so that the regular CAL actors shown in Figure 1 can be executed according to the quasi-static scheduling method that was described in this article. The work is still somewhat in progress, but the quasi-statically scheduled execution has already shown a run-time performance advantage over the unscheduled execution of CAL actors on a workstation.

Our quasi-static scheduling approach is capable of producing multiprocessor schedules. However, verifying the functionality of the multiprocessor schedules requires a multiprocessing system that supports the execution of the RVC decoder. Establishing such a system...
is a demanding task that requires a considerable amount of work. Nevertheless, stand-alone experiments to verify the functionality of EPFS run-time scheduling have been conducted on an FPGA platform that runs several soft cores according to a quasi-static EPFS schedule.

7 Discussion

The presented quasi-static scheduling method greatly reduces the amount of run-time decision making. In the original CAL model, starting of almost every EFSM action requires checking one or more control conditions. With our scheduling, the number of these checks is reduced to one per system-level graph. In numbers, this means that for the CAL network part that is scheduled quasi-statically, the number of action transition checks is reduced approximately to \( \frac{1}{300} \) (See Table 2) of the unscheduled CAL model.

The preprocessing steps (identifying \( P \) and token rates) 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. Experiments have shown that the proposed method offers a performance advantage over un-scheduled actor execution.

Acknowledgements This research has been partially funded by the Nokia Foundation, Finnish Graduate School for Electronics, Telecommunication and Automation, and the Tekes project ECUUS. The authors would like to thank the reviewers for comments that helped improving this article.

References


