A Rule-Based Static Dataflow Clustering Algorithm for Efficient Embedded Software Synthesis

Joachim Falk, Christian Zebelein, Christian Haubelt, and Jürgen Teich
Department of Computer Science, University of Erlangen-Nuremberg
Email: {falk, christian.zebelein, haubelt, teich}@cs.fau.de

***Abstract*—In this paper, an efficient embedded software synthesis approach based on a generalized clustering algorithm for *static dataflow subgraphs* embedded in general dataflow graphs is proposed. The clustered subgraph is *quasi-statically scheduled*, thus improving performance of the synthesized software in terms of latency and throughput compared to a dynamically scheduled execution. The proposed clustering algorithm outperforms previous approaches by a faster computation and a more compact representation of the derived quasi-static schedules. This is achieved by a *rule-based* approach, which avoids an explicit enumeration of the state space. Experimental results show significant improvements in both performance and code size when compared to a state-of-the-art clustering algorithm.**

***Index Terms*—MPSoC Scheduling, Software Synthesis, Actor-Oriented Design**

I. Introduction

For the implementation of complex multimedia and signal processing applications, Multi-Processor System on Chip (MPSoC) architectures are becoming more and more important. However, due to the high degree of parallelism, programming these MPSoCs with traditional programming languages becomes quite error prone [1]. Fortunately, *dataflow graphs*, which have been applied successfully to application modeling in these domains, expose inherent application parallelism, and thus, map well into the MPSoC world. Thus, in the following, it is assumed that the application is given as a dataflow graph. Nowadays, typical multimedia applications can neither be modeled by purely static nor fully dynamic dataflow graphs. In such scenarios, static dataflow subgraphs are embedded in more general (dynamic) dataflow graphs. Hence, an embedded software synthesis approach has to support heterogeneous dataflow graphs.

A *dataflow graph* is a directed graph $g = (A, C)$, where the set of vertices A represents the *actors* and the set of edges $C \subseteq A \times A$ represents the *channels*. Actors in a dataflow graph perform the actual computation by so called *firings*. An actor can *fire* if a sufficient number of tokens is available on its input channels (incoming edges). When an actor fires, it consumes tokens from its input channels and produces tokens onto its output channels (outgoing edges). The behavior of an actor might be either *static* or *dynamic*: Whereas static actors consume and produce tokens with constant rates (synchronous dataflow (SDF), [2]) or periodically constant rates (cyclo-static dataflow (CSDF), [3]), dynamic actors have variable rates.

Once the mapping of the actors onto the MPSoC has been determined, a major problem in embedded software synthesis is to efficiently schedule the actors bound onto the same resource. One option is use a dynamic scheduler. In this case, the actors are checked at runtime for executability following the chosen scheduling policy, e.g., in a round-robin fashion, and executable actor are fired by the scheduler. However, dynamic scheduling can be detrimental to system performance, especially if the scheduled actors are very fine grained. In this case, the scheduling overhead incurred by runtime decisions takes up a noticeable amount of the total computational time. This is especially true if the multimedia or signal processing application contains parts that can be scheduled statically, which is often the case.

As *static schedules* do not require any runtime decisions, statically scheduling these parts seems to be another option. A static schedule is a sequence of actor firings τ, which is periodically executed, and can be derived from the so called *repetition vector* $\boldsymbol{\mu} = (\mu_{a_1}, \mu_{a_2}, \ldots \mu_{a_m})$ of the static dataflow graph. Given such a schedule τ, each actor $a_i \in A$ is fired exactly μ_{a_i} times. However, as the resulting subsystem has to communicate with dynamic parts of the application, computing static schedules is not viable in the general case.

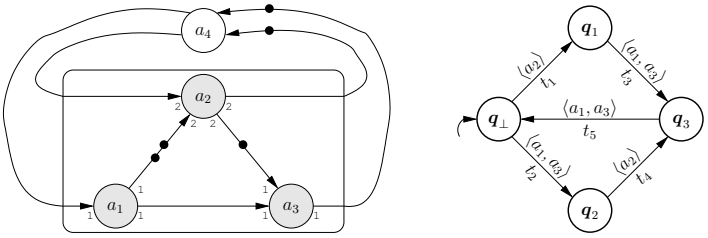
The scheduling overhead problem could be mended by choosing an appropriate level of granularity, i.e., merging as much functionality into a single actor such that the computation costs dominate the scheduling overhead. However, such a merging step is equivalent to *manually clustering* the dataflow graph, i.e., static dataflow actors are clustered into a single composite actor, which is then scheduled for a single processing core.

In this paper, an approach that clusters static dataflow actors into a single *composite actor* implementing a so-called *quasi-static schedule* (QSS) is proposed. In a QSS, runtime decisions are followed by statically scheduled sequences of actor firings. However, generating too long statically scheduled sequences may introduce deadlocks. To prevent this, clustering assumes that the environment exhibits worst case behavior, so-called *tight feedback loops*, which occur if each token produced by the composite actor in one step is again required to produce the input to execute an immediately following step of the composite actor.

For example, consider the dataflow graph shown in Fig. 1a with static actors a_1, a_2 and a_3. The repetition vector of the induced static dataflow subgraph is $\boldsymbol{\mu} = (2, 1, 2)$. A possible fully static single processor schedule for this subgraph is $\langle a_2, a_1, a_3, a_1, a_3 \rangle$. In order to execute this static schedule atomically (otherwise it would not be a static schedule), at least two tokens must be available on each input channel (a_4, a_1) and (a_4, a_2). Assuming that the dynamic actor a_4 is conservative, i.e., the number of produced and consumed tokens per firing is identical, this firing rule is never satisfied as only two tokens are present to be forwarded by a_4, while the static schedule requires four tokens to be executed. Thus, implementing a static schedule might introduce deadlocks into the clustered system.

On the other hand, a QSS that does not introduce deadlocks is shown in Fig. 1b. The QSS is represented by a finite state machine

978-3-9810801-7-9/DATE11/©2011 EDAA



a) Heterogeneous dataflow graph b) Quasi-static schedule

Fig. 1. Heterogeneous dataflow graph with a single dynamic dataflow actor a_4 and three static dataflow actors a_1, a_2, a_3. The constant consumption and production rates of the static dataflow actors are annotated to the edges. The static dataflow actors can be clustered into a single actor reducing the scheduling overhead. The resulting quasi-static schedule is represented by an FSM with four states $\boldsymbol{q}_\perp$, $\boldsymbol{q}_1$, $\boldsymbol{q}_2$, $\boldsymbol{q}_3$ and five transitions $t_1, \ldots, t_5$. Static schedule sequences are annotated to the transitions.

(FSM), with edges representing static scheduling sequences. Depending on the forwarding decision of the dynamic actor a_4, either $\langle a_2 \rangle$ (two tokens on (a_4, a_2)) or $\langle a_1, a_3 \rangle$ (one token on (a_4, a_1)) can be fired. Note that this will produce new tokens on the cluster outputs for actor a_4 to forward, thus keeping the system running.

This paper presents an improvement of clustering synthesis starting from a given set of actors to cluster. In contrast to previous embedded software synthesis approaches based on clustering, the enumeration of the state space is avoided. For this purpose, state sets are represented *implicitly* and state transitions are defined by *rules*. Hence, the proposed clustering approach is able to schedule larger systems and generates more compact schedules. The latter results in considerable code size reductions of the generated software.

The remainder of this paper is organized as follows: Related work will be reviewed in Section II. Section III presents the proposed embedded software synthesis approach based on a novel clustering algorithm. Experimental results presented in Section IV will show the performance improvements by applying the proposed clustering algorithm to heterogeneous dataflow graphs. Moreover, a comparison to a state-of-the-art clustering approach will show the efficiency of the proposed clustering algorithm in terms of scalability and code size reduction.

II. Related Work

A good overview on model-based software design flows for MPSoCs is given in [4]. When mapping a dataflow application onto an MPSoC platform, the overall performance of the implementation is not only highly dependent on the actual mapping of actors to processing cores ([5], [6], [7], [8], [9]), but also depends on the scheduling on each individual processing core. For instance, efficient single processor scheduling algorithms [10], [11], [12] exist for *synchronous dataflow (SDF)* graphs [2], a widely accepted static dataflow model.

Unfortunately, purely static modeling approaches are insufficient to model real world multimedia and signal processing applications. Moreover, simply applying these static scheduling methodologies to static subgraphs embedded in general dataflow graphs introduces deadlocks into the resulting system in the general case. However, the advantages of clustering SDF subgraphs for the purpose of generating static schedules have been shown in [13], [14], which introduced the *Acyclic Pairwise Grouping of Adjacent Nodes (APGAN)* algorithm for constructing lexical orderings for later conversion into *single appearance schedules*. APGAN could in principle be used for clustering heterogeneous dataflow systems due to the restriction that only acyclic graphs are handled. This evades the problem of feedback loops as considered in the proposed approach in the paper at hand.

Process merging while keeping throughput constraints has been explored by Stefanov et al. [15]. However, the presented methodology is only applicable to acyclic graphs, thus, the deadlock problem handled in this paper does not appear. Furthermore, these acyclic graphs are not arbitrary acyclic graphs but graphs induced by nested loop programs.

Plishker et al. [16], [17] have presented a scheduling methodology for dynamic dataflow graphs which improves on the simple round-robin scheduler. The actors in this dynamic graphs are constrained to switch between static modes. If an SDF subgraph exhibits multirate behavior, an optimized schedule is constructed, which checks actors for activatability in correct proportion to each other. Plishker's approach is more general, as it can handle certain kinds of dynamic actors, but also more limited as it still needs to check the activatability of each actor firing. In contrast, the approach presented here can detect whole chains of actor firings which can be executed statically.

Finally, in [18] the deadlock problem due to tight feedback loops has been handled. However, schedules are represented by FSMs, which has some severe disadvantages. Looking again at the QSS shown in Fig. 1b, it can be observed that transitions t_1 and t_4 as well as t_2 and t_3 have the same actor firings. Moreover, the state sets can be described more succinctly by the rules presented in this paper, whereas the explicit enumeration of states generally leads to huge schedule representations.

The proposed approach in this paper has two major contributions: (1) the *construction* of a QSS for clustered static dataflow subgraphs is *faster* than the approach proposed in [18] due to an implicit representation of the states in the QSS, and (2) the generated QSS generally leads to considerably more *compact source code* when compared to the approach in [18] due to a rule-based representation of the quasi-static schedule.

III. Clustering of Static Dataflow Subgraphs

The clustering approach presented in this section replaces a given static dataflow subgraph $g_\mathrm{S} = (A_S, C_S)$ by a composite actor a_c based on the computation of a quasi-static schedule (QSS) for a_c. This is done in such a way that a_c does not introduce deadlocks into the system, assuming that the original system is deadlock-free.

The proposed algorithm works on the set of cluster *input* and *output actors*, $A_\mathrm{I} \subseteq A_S$ and $A_\mathrm{O} \subseteq A_S$, respectively. Input actors are static actors having at least one incoming channel from an actor outside the cluster, while output actors are static actors having at least one outgoing channel to an actor outside the cluster. Note that A_I and A_O may not be disjoint sets. For example, the static dataflow subgraph from Fig. 1 with $A_S = \{a_1, a_2, a_3\}$ has two input and two output actors, i.e., $A_\mathrm{I} = \{a_1, a_2\}$ and $A_\mathrm{O} = \{a_2, a_3\}$. Note that a static dataflow subgraph can only be clustered by the proposed method if for each pair of input and output actors $(a_\mathrm{i}, a_\mathrm{o}) \in A_\mathrm{I} \times A_\mathrm{O}$ a directed path $p \in C^*$ from actor a_i to actor a_o exists (the so-called *clustering condition*, cf. [18]).

The proposed clustering approach basically works in two steps: In the first step, the state space given by static dataflow actor firings is implicitly constructed using a *rule-based approach*. The result of the first step are rules that specify possible actor firings

TABLE I
INPUT/OUTPUT DEPENDENCY FUNCTION VALUES FOR SUBGRAPH g_S FROM FIG. 1 AND CORRESPONDING INPUT/OUTPUT DEPENDENCY STATES.

$\mathbf{dep}_{\mathrm{io}}(a_o, n) = (q_{a_1}, q_{a_2})$			Q_{io}	
n	$a_o = a_2$	$a_o = a_3$	$\boldsymbol{q} = (q_{a_1}, q_{a_2}, q_{a_3})$	
0	$(0,0)$	$(0,0)$	$(0,0,0)$	$(0,0,0)$
1	$(0,1)$	$(1,0)$	$(0,1,0)$	$(1,0,1)$
2	$(2,2)$	$(2,1)$	$(2,2,2)$	$(2,1,2)$
...	...	...	...	...

in the static dataflow subgraph in dependence of previous actor firings. Note that the possible actor firings are still unscheduled. Hence, in a second step, a feasible single processor schedule is computed resulting in static actor firing sequences that don't require any dynamic scheduling decisions.

A. Implicit State Space Representation

The proposed algorithm operates on the so-called *input/output dependency function* $\mathbf{dep}_{\mathrm{io}} : A_{\mathrm{O}} \times \mathbb{N}_0 \to {\mathbb{N}_0}^{|A_{\mathrm{I}}|}$, which is a vector-valued function that associates with each output actor $a_o \in A_{\mathrm{O}}$ and a given number of firings $n \in \mathbb{N}_0$ the minimum number of required input actor firings $(q_{a_{i,1}}, q_{a_{i,2}}, \ldots, q_{a_{i,|A_{\mathrm{I}}|}})$. With this information, it can be calculated how many tokens are required on the input channels in order to produce output tokens. Consequently, an actor can be fired as soon as the specified amount of tokens is available. Hence, to perform a requested number n of firings of a given output actor $a_o \in A_{\mathrm{O}}$, at least $\mathbf{dep}_{\mathrm{io}}(a_o, n)$ input actor firings are required. Consider again the dataflow graph given in Fig. 1: In order to fire actor a_3 once, $\mathbf{dep}_{\mathrm{io}}(a_3, 1) = (1, 0)$ input actor firings are required, i.e., actor a_1 has to be fired once. From the consumption rate of actor a_1, it is known that in this case one token on input channel (a_4, a_1) is required. However, in order to fire actor a_3 twice, $\mathbf{dep}_{\mathrm{io}}(a_3, 2) = (2, 1)$ input actor firings are required, i.e., actor a_1 must be fired twice and actor a_2 must be fired once. Further values are depicted in Table I. Note that while this table is infinite it is also periodic with respect to the repetition vector $\boldsymbol{\mu}_{\mathrm{io}}$, e.g., $\boldsymbol{\mu}_{\mathrm{io}} = (2, 1, 2)$ for the example in Table I where $(2, 2, 2)$ is an equivalent value for $(0, 1, 0)$ and $(2, 1, 2)$ for $(0, 0, 0)$. The periodicity can be exploited to only work on the finite number of values in a period.

Given $\mathbf{dep}_{\mathrm{io}}$, the so-called *input/output dependency states* Q_{io} can be calculated. Each input/output state $\boldsymbol{q} = (q_{a_1}, q_{a_2}, \ldots, q_{a_m}) \in Q_{\mathrm{io}} \subset {\mathbb{N}_0}^{|A_{\mathrm{I}} \cup A_{\mathrm{O}}|}$ represents a possible state of the cluster in a self scheduled execution, and is additionally constrained to provide the maximum possible number of output actor firings for a minimum number of required input actor firings. This constraint stems from the fact that the clustering algorithm has to assume a worst case behavior of the subgraph's environment, and thus, must neither consume more input tokens than necessary nor postpone the production of output tokens.

Due to the definition of $\mathbf{dep}_{\mathrm{io}}$, the number of input actor firings from the input/output dependency function $\mathbf{dep}_{\mathrm{io}}$ are already minimal. Therefore, in order to derive an input/output dependency state from $\mathbf{dep}_{\mathrm{io}}$, only the output actor firings have to be maximized, as defined below:[1]

$$
\begin{aligned}
&Q_{\mathrm{io}} = \{\boldsymbol{q}_\perp\} \cup \{(q_{a_1}, q_{a_2}, \ldots q_{a_m}) \mid a \in A_{\mathrm{O}}, n \in \mathbb{N}_0, \\
&\forall a_o \in A_{\mathrm{O}} : q_{a_o} = \max\{k \in \mathbb{N}_0 \mid \mathbf{dep}_{\mathrm{io}}(a_o, k) \leq \mathbf{dep}_{\mathrm{io}}(a, n)\} \\
&\forall a_i \in A_{\mathrm{I}} : q_{a_i} = \mathbf{dep}_{\mathrm{io}}(a, n)_{a_i}\}
\end{aligned}
$$

[1]In the following, comparisons between two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ will be defined as follows: $\boldsymbol{a} \leq \boldsymbol{b} \iff \forall i : a_i \leq b_i$ and $\boldsymbol{a} < \boldsymbol{b} \iff \boldsymbol{a} \leq \boldsymbol{b} \wedge \boldsymbol{a} \neq \boldsymbol{b}$.

The initial state $\boldsymbol{q}_\perp$ is the all zero vector representing the fact that in the beginning, no actor of the subgraph has been fired. Note that the maximum operation calculates for each output actor a_o the maximum number of firings which can be performed by at most $\mathbf{dep}_{\mathrm{io}}(a, n)$ firings of input actors. This can be done efficiently, as the values of $\mathbf{dep}_{\mathrm{io}}$ for a given output actor a_o are totally ordered under $\leq$, i.e., $\mathbf{dep}_{\mathrm{io}}(a_o, n) \leq \mathbf{dep}_{\mathrm{io}}(a_o, n+1)$. For the subgraph of Fig. 1, Q_{io} is shown in Table I (fourth and fifth column).

As the calculation of the input/output dependency states Q_{io} considers each output actor individually, Q_{io} does not contain states resulting from different interleavings of actor firings that are permitted by the partial order of these actor firings. These interleavings are captured by the *state space* Q, which is defined as the least fixpoint $Q = \mathbf{lfp}(Q' = \{\max(\boldsymbol{q}_1, \boldsymbol{q}_2) \mid \boldsymbol{q}_1, \boldsymbol{q}_2 \in Q'\} \cup Q' \cup Q_{\mathrm{io}})$ which enlarges Q' starting from Q_{io} by adding the pointwise maximums of all pairs of input/output states from Q' until no more new states are created. However, to avoid this explicit enumeration of the state space Q, *rules* $r \in R$ will be used instead. A rule maps a subspace Q_r of the state space Q to a vector of actor firings $\boldsymbol{s}$ (in the following called *partial repetition vector*) that can be executed if the current state is element of Q_r.

Definition III.1 (Rule) *A rule is a tuple $r = (\boldsymbol{l}, \boldsymbol{u}, \boldsymbol{s})$. The vectors $\boldsymbol{l} \in {\mathbb{N}_0}^{|A_{\mathrm{I}} \cup A_{\mathrm{O}}|}$ and $\boldsymbol{u} \in {\mathbb{N}_{0,\infty}}^{|A_{\mathrm{I}} \cup A_{\mathrm{O}}|}$, $\boldsymbol{l} \leq \boldsymbol{u}$, define the subspace of the state space $Q_r \subseteq Q$ where the rule is active, i.e., $Q_r = \{\boldsymbol{q} \in Q \mid \boldsymbol{l} \leq \boldsymbol{q} \leq \boldsymbol{u}\}$.[2] The partial repetition vector $s \in {\mathbb{N}_0}^{|A_{\mathrm{I}} \cup A_{\mathrm{O}}|}$ specifies for each input/output actor how many firings to perform when r is executed.*

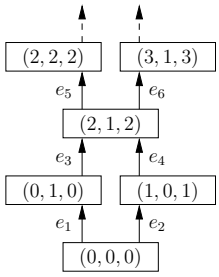
B. Derivation of Rules

The first step of the proposed clustering approach is to find a set of initial rules *rules* R_{ini} based on the input/output dependency states Q_{io}. Basically, the elements of R_{ini} correspond to the edges of the Hasse diagram induced by the partial order of the elements in Q_{io} under $\leq$ (cf. Fig. 2a). Given an edge $\boldsymbol{q}_1 \to \boldsymbol{q}_2$ in this diagram, we know that $\boldsymbol{q}_1 < \boldsymbol{q}_2$ and $\nexists \boldsymbol{q} \in Q_{\mathrm{io}} : \boldsymbol{q}_1 < \boldsymbol{q} < \boldsymbol{q}_2$.

Obviously, the rule r generated by such an edge has to perform $\boldsymbol{s} = \boldsymbol{q}_2 - \boldsymbol{q}_1 > \boldsymbol{0}$ actor firings. The lower bound $\boldsymbol{l}$ of r is equal to $\boldsymbol{q}_1$, representing the minimum number of previous actor firings in order to enable the rule. The upper bound $\boldsymbol{u}$ of r can be derived as follows: If an actor a is fired by r, i.e., $s_a > 0$, the exact number of previous firings of a has to be known. This is expressed by setting $u_a = l_a$. This constraint stems from the fact that the availability of sufficient tokens in order to execute a s_a times can only be guaranteed for the lower bound l_a. If an actor a is not fired by r, i.e., $s_a = 0$, only the minimum number of previous firings of a has to be ensured. This is expressed by setting $u_a = \infty$. This constraint ensures that at least the minimum number of tokens have been produced by a. Note that if more firings of a have already been performed than indicated by l_a, a must also have produced more tokens than required, not less.

For example, the Hasse diagram corresponding to the input/output dependency states from Table I is shown in Fig. 2a. The rule r_3 corresponding to edge e_3 is then derived as follows: $\boldsymbol{s} = (2, 1, 2) - (0, 1, 0) = (2, 0, 2)$, $\boldsymbol{l} = (0, 1, 0)$, and $\boldsymbol{u} = (0, \infty, 0)$. Therefore, in order to enable the rule, actors a_1 and a_3 must not have been fired before, whereas a_2 must have been

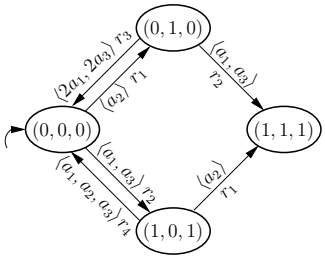
[2]$\mathbb{N}_{0,\infty} = \mathbb{N}_0 \cup \{\infty\}$ denotes the set of non-negative integers including infinity.



a) Hasse diagram

r	e	$\boldsymbol{l}$	$\boldsymbol{u}$	$\boldsymbol{s}$
r_1	e_1	$(0,0,0)$	$(\infty,0,\infty)$	$(0,1,0)$
r_2	e_2	$(0,0,0)$	$(0,\infty,0)$	$(1,0,1)$
r_3	e_3	$(0,1,0)$	$(0,\infty,0)$	$(2,0,2)$
r_4	e_4	$(1,0,1)$	$(1,0,1)$	$(1,1,1)$
r_5	e_5	$(2,1,2)$	$(\infty,1,\infty)$	$(0,1,0)$
r_6	e_6	$(2,1,2)$	$(2,\infty,2)$	$(1,0,1)$
...	...	...	...	...

b) Initial rules R_{ini}



c) Simulated FSM from $R_{\min}$

r	$\boldsymbol{l}$	$\boldsymbol{u}$	$\boldsymbol{s}$
r_1	$(0,0,0)$	$(\infty,0,\infty)$	$(0,1,0)$
r_2	$(0,0,0)$	$(0,\infty,0)$	$(1,0,1)$
r'_3	$(1,1,1)$	$(1,\infty,1)$	$(1,0,1)$

d) Final rules R_{fin}

Fig. 2. a) Hasse diagram corresponding to the input/output dependency states from Table I, b) rules derived from the edges of the Hasse diagram, c) FSM resulting from simulating rules $R_{\min}$ (applied rules and actor firings are annotated to the edges), and d) final rules derived from the rules in (b) through conflict resolving.

fired once (or more). Further rules are given in Fig. 2b. Similar to Table I this table is infinite but periodic, e.g., rules r_1 and r_5 as well as r_2 and r_6 are equivalent with respect to the repetition vector $\boldsymbol{\mu}_{\text{io}} = (2,1,2)$.

In principle, the set of rules can be simulated in order to obtain an FSM corresponding to these rules: Starting from state $\boldsymbol{q} = \mathbf{0}$, each rule's subspace Q_r is compared to the current state $\boldsymbol{q}$. If $\boldsymbol{q} \in Q_r$ then rule r is selected for execution. The destination state $\boldsymbol{q}''$ is then calculated in two steps: First, the number of actor firings specified by $r.\boldsymbol{s}$ are added to the current state, i.e., $\boldsymbol{q}' = \boldsymbol{q} + r.\boldsymbol{s}$. Then, $\boldsymbol{q}'' = \boldsymbol{q}' \bmod \boldsymbol{\mu}_{\text{io}}$, where $\boldsymbol{\mu}_{\text{io}}$ is the projection of the cluster's repetition vector $\boldsymbol{\mu}$ onto input and output actors.[3] The simulation is finished when no more new states are discovered.

Note that the the modulo operation is permitted due to the fact that the state of the cluster is equivalent before and after executing a schedule τ corresponding to the repetition vector $\boldsymbol{\mu}$. Due to this fact, it is sufficient to only consider a subset of rules $R_{\min} \subset R_{\text{ini}}$ to simulate the FSM. This set of rules is only responsible for states $\boldsymbol{q}$ in the first period $\mathbf{0} \leq \boldsymbol{q} \not\geq \boldsymbol{\mu}_{\text{io}}$ of the infinite state space Q.

For example, simulating the set of rules $R_{min} = \{r_1, r_2, r_3, r_4\}$ (cf. Fig. 2b) results in the FSM depicted in Fig. 2c. In contrast to the FSM shown in Fig. 1b, it can be observed that the state $(1,1,1)$ has no outgoing transitions. This is due to the fact that the FSM generated by the rules is non-deterministic: If the current state of the FSM is $(0,1,0)$ and there are enough tokens to execute r_2, the next state is $(1,1,1)$. When enough tokens are available to fire a_1 and a_3 for the second time (as done by transition $\boldsymbol{q}_3 \rightarrow \boldsymbol{q}_\perp$ in Fig. 1b), a backtracking to state $(0,1,0)$ is required where a_1 and a_3 are fired twice (by rule r_3). Obviously, such a backtracking approach is unrealistic for dataflow graphs, as actors cannot be "unfired", thus leading to deadlock. This is caused by so-called *conflict rules*, which can, however, be resolved by adding additional rules, e.g., adding a rule which can transition from $(1,1,1)$ to $(0,0,0)$.

C. Resolving Conflict Rules

The set of deadlock-free rules R is calculated by the least fixpoint operation $R = \mathbf{lfp}(R' = \{r'_1, r'_2 \mid r_1, r_2 \in R' \wedge r_1 \text{ and } r_2 \text{ are in conflict}\} \cup R' \cup R_{\text{ini}})$ which enlarges R' starting from R_{ini} by adding the rules r'_1 and r'_2 calculated by the conflict resolution of the two conflicting rules r_1 and r_2 until no more new rules are created.

Two rules r_1 and r_2 are said to be in conflict if both have at least one common state $\boldsymbol{q}$ in which they can be activated and fire at least one common actor. Hence, as both rules are enabled in state $\boldsymbol{q}$ (assuming enough tokens are available), executing either rule disables the other rule (proof omitted due to space constraints). Note that if two rules have no common state, they cannot be in conflict, as they can never be enabled at the same time. Also, if two rules have indeed a common state $\boldsymbol{q}$, but fire no common actors, they are not in conflict either.

In order to resolve such a conflict situation, new rules will be added to R: For each conflicting rule pair r_1 and r_2, common actor firings are extracted from both rules, resulting in two additional rules r'_1 and r'_2. Then, r'_1 can be applied after r_2, and analogously, r'_2 can be executed after r_1 (proof omitted due to space constraints). Let $\boldsymbol{s}_c = \min(r_1.\boldsymbol{s}, r_2.\boldsymbol{s})$ be the pointwise minimum of the partial repetition vectors $r_1.\boldsymbol{s}$ and $r_2.\boldsymbol{s}$ representing the common actor firings of r_1 and r_2. Then, the lower bounds and partial repetition vectors of r'_1 and r'_2 can be calculated as follows:

$$r'_1.\boldsymbol{l} = r_1.\boldsymbol{l} + \boldsymbol{s}_c \qquad r'_1.\boldsymbol{s} = r_1.\boldsymbol{s} - \boldsymbol{s}_c$$
$$r'_2.\boldsymbol{l} = r_2.\boldsymbol{l} + \boldsymbol{s}_c \qquad r'_2.\boldsymbol{s} = r_2.\boldsymbol{s} - \boldsymbol{s}_c$$

The upper bounds $r'_1.\boldsymbol{u}$ and $r'_2.\boldsymbol{u}$ are then calculated from these lower bounds and partial repetition vectors as described in Section III-B. A special case, which leads to the marking of rules as *redundant* during conflict resolving, arises if for a pair of rules r_1 and r_2, $r_1.\boldsymbol{s} > r_2.\boldsymbol{s}$ and $Q_{r_1} \subseteq Q_{r_2}$. Then r_1 is a redundant rule (proof omitted due to space constraints). An analogous observation applies to rule r_2. Redundant rules are added to the R_{red} set and are removed in the final post-processing step to calculate the final rules R_{fin}. A resulting rule r'_1 or r'_2 with zero partial repetition vector, i.e., $\boldsymbol{s} = \mathbf{0}$, is discarded and not added to R, as it does not fire any actors.

Considering the set of rules $R_{min} = \{r_1, r_2, r_3, r_4\}$ from Fig. 2b, a pair of conflicting rules is, e.g., r_2 and r_3: A common state of these rules is, e.g., $\boldsymbol{q} = (0,1,0)$, and the vector of common actor firings is $\boldsymbol{s}_c = (1,0,1) > \mathbf{0}$. As a result, applying the conflict resolving operation to r_2 and r_3 results in two rules: $r'_2 = ((1,0,1), (\infty,\infty,\infty), (0,0,0))$ (discarded due to the zero partial repetition vector), and $r'_3 = ((1,1,1), (1,\infty,1), (1,0,1))$. Note that r'_3 is exactly the missing rule which creates the transition between states $(1,1,1)$ and $(0,0,0)$ in Fig. 2c. The other pair of conflict rules, r_1 and r_4, does not introduce any other new rules, as $r'_4 = ((1,1,1), (1,\infty,1), (1,0,1)) = r'_3$ in this case. Note that r_3 and r_4 have been marked as redundant in the process, resulting in the final set of rules as shown in Fig. 2d. Compared to the FSM from Fig. 1b, we can observe that transitions t_1 and t_4 are created by r_1, t_2 and t_3 by r_2, and t_5 by r'_3.

[3] We define the modulo operation between two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ such that $\boldsymbol{a} \bmod \boldsymbol{b}$ denotes the smallest vector $\boldsymbol{a}' = \boldsymbol{a} - m \cdot \boldsymbol{b}, m \in \mathbb{N}_0$ such that $\boldsymbol{a}' \geq \mathbf{0}$.

To calculate R_{fin} we remove the redundant rules R_{red} from R and project all remaining rules into the first period, i.e., $R_{\text{fin}} = \{(\boldsymbol{l} - m \cdot \boldsymbol{\mu}_{\text{io}}, \boldsymbol{u} - m \cdot \boldsymbol{\mu}_{\text{io}}, s) \mid (\boldsymbol{l}, \boldsymbol{u}, \boldsymbol{s}) \in R - R_{\text{red}}, \text{ where } m = \boldsymbol{l} \operatorname{\mathbf{div}} \boldsymbol{\mu}_{\text{io}}\}$. [4] Finally, for each rule $r \in R_{\text{fin}}$, its actor firings as specified by the partial repetition vector $\boldsymbol{s}$ are scheduled by a modified version of the cycle-breaking algorithm presented in [11], which, for pure SDF graphs, always finds a single appearance schedule (SAS) if one exists.

IV. Results

In order to illustrate the benefits of our clustering algorithm developed in Section III, we have applied it to both, synthetic dataflow graphs as well as an mp3 decoder (cf. Fig. 3), which works on the *mp3 granule level* (i.e., 576 frequency/time domain samples). For two subgraphs QSSs have been computed using our proposed approach. To evaluate the scheduling overhead reduction in isolation, we first removed as much functionality as possible from all actors. The dynamic scheduler for this mp3 decoder required about 184 ms runtime for a given mp3 input stream (approx. 20 MB). This decreases to 57 ms when using the previously computed QSSs for the two subgraphs, i.e., an improvement of approx. 69%. For a real world test we switched back to the unmodified mp3 decoder. This resulted in about 2,230 ms decoding time for the same input stream using the dynamic scheduler, and 2,100 ms when using the QS schedules. This corresponds to an improvement of approx. 6%. However, for dataflow graphs with more fine grained actors, the achievable scheduling overhead reduction is expected to be higher.

In order to evaluate the methodology more thoroughly, we also applied it to 400 randomly generated dataflow graphs: Using the SDF3 tool [19], we generated four different SDF graphs G2, G3, G5, and G7 with 60 actors each. The generated graphs are cyclic, i.e., they contain strongly connected components, with random but consistent SDF rates and sufficient initial tokens in order to guarantee a deadlock-free self-scheduled execution. All four graphs have the same properties, except the degree of connectivity, i.e., the number of edges. The average input/output degree of each actor in G2 is 2, in G3 it is 3, in G5 it is 5, and in G7 it is 7. Based on these four graphs, test cases G2-N_D, G3-N_D, G5-N_D, and G7-N_D have been constructed, by randomly marking a variable number N_D of actors as dynamic. For each test case, the settings $N_D = 6, 12, 18, \ldots, 54, 60$ have been considered, and for each setting, ten instances have been generated, i.e., G2-06-01, G2-06-02, ... G7-60-10. The ten different instances per setting are used to compute average results in the experiments.

[4] We define the div operation between two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ such that $\boldsymbol{a} \operatorname{\mathbf{div}} \boldsymbol{b}$ denotes the largest integer m such that $\boldsymbol{a} - m \cdot \boldsymbol{b} \geq \boldsymbol{0}$.

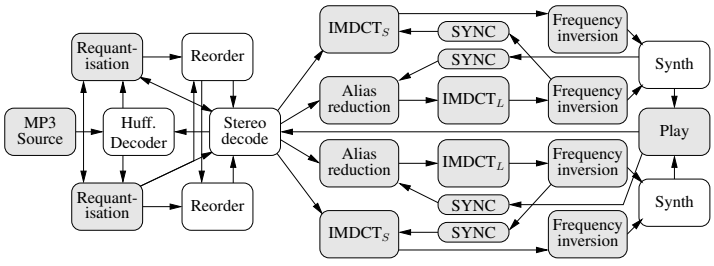


Fig. 3. Dataflow graph of a mp3 decoder. Shaded vertices correspond to static actors, which are grouped into four subgraphs (two subgraphs consist of a single actor only). The remaining two subgraphs are processed by the proposed scheduling method to compute their QSS schedules.

A first observation is that the size of the clusters satisfying the cluster condition depends not only on the number of dynamic actors in the graph but also on the degree of connectivity. This is illustrated in Fig. 4a. It can be seen that a lower degree of connectivity leads to a smaller number of static actors that can be clustered. Hence, the test case G2-N_D has on average smaller clusters than the test case G3-N_D for the same N_D. G3-N_D again has on average smaller clusters than G5-N_D, and so on. A further result is that test cases with N_D greater than or equal to 30 dynamic actors can be neglected. This is due to the fact that by randomly marking actors as dynamic, the more dynamic actors exist, the smaller the clusters become, i.e, in the worst case the remaining n static data flow actors are evenly scattered in the graph, eventually resulting in n clusters of size one.

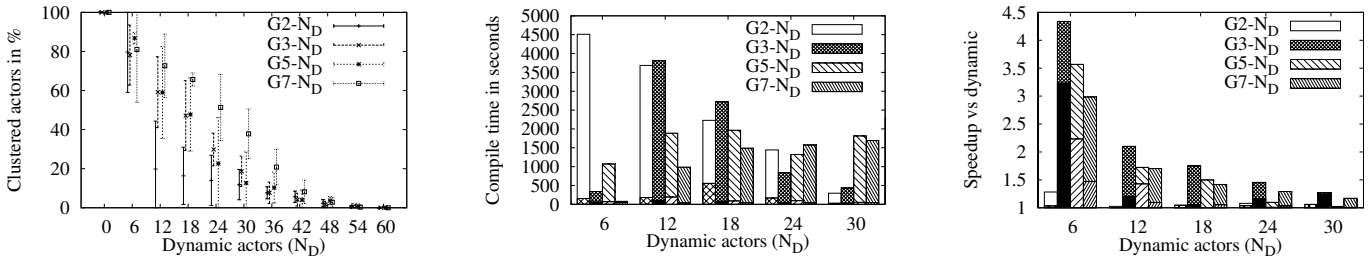
Next, for all the test cases we applied our proposed rule-based clustering for computing quasi static schedules and used them during software synthesis. In order to identify clusters that satisfy the cluster condition, we use a greedy algorithm based on a SAT-solver engine. Note that there are generally many different clusterings for a single graph. In future work, we will use this fact to improve our results by an automatic optimization.

Using the greedy algorithm, software has been generated by (M1) fully dynamically scheduling the dataflow graph (as reference), (M2) our proposed rule-based approach, and (M3) the FSM approach proposed in [18]. The achieved average speedup (SU) by M2 and M3 compared to M1 and its standard deviation as well as the average code size (CS) of the binary and its standard deviation for both clustering approaches is shown in Table II. Better values are printed in bold. It can be seen that in all cases the code size generated by our proposed approach (M2) is smaller than the code size of the software produced by the FSM approach (M3). In some cases, the code size is even smaller than the code size of the dynamically scheduled software (M1).

For the speedup it can be observed that the FSM approach (M3) produces better results for dataflow graphs with low connectivity and for graphs with few dynamic actors. For the harder cases, our proposed approach often produces faster code. However, the performance might drop to the performance of the dynamic scheduler in very hard cases.

Our proposed rule-based clustering approach (M2) has another advantage when comparing it with the FSM approach (M3) on the basis of the required compile time. In these results, M2's compile time is always a small fraction of M3's compile time (orders of magnitude). This significantly extends the area of applicability of clustering during embedded software synthesis.

In embedded systems, the available amount of memory is typically constrained. Hence, we tested both clustering approaches with given code size constraints. The code size constraints have been set to 133% of the memory requirements of the previously computed dynamic schedules (M1). The average speedup is shown in Fig. 4c. It can be seen that in all cases the proposed rule-based approach outperforms the FSM approach. Even more, while the FSM approach is only able to produce moderate speedup for the easy test cases ($N_D = 6$ and $N_D = 12$), our proposed rule-based approach is even able to improve the throughput in many hard cases $N_D > 12$).



a) Cluster size b) Compile time c) Speedup 133% code size

Fig. 4. a) Average number and standard deviation of static actors that can be clustered. b) Average compile time (Core^{TM}2 Quad 3 GHz) of the FSM approach (M3) and the rule-based approach (M2), which is always a small fraction of M3's compile time. c) Average speedup respecting code size constraints (133% of the memory required by the dynamic schedule). Speedup of the FSM approach is always less than the speedup of the rule-based approach.

TABLE II
AVERAGE SPEEDUP, CODE SIZE, AND THEIR STANDARD DEVIATIONS

test case		M1	N_D = 6		N_D = 12		N_D = 18		N_D = 24		N_D = 30	
		CS[kB]	SU	CS[kB]	SU	CS[kB]	SU	CS[kB]	SU	CS[kB]	SU	CS[kB]
G2-N_D	M2	18	1.74±0.87	**27±6**	1.03±0.06	**19±2**	1.05±0.06	**18±0**	**1.08±0.11**	**18±0**	1.04±0.04	**18±0**
	M3	18	**3.31±2.27**	286±495	**1.16±0.47**	138±360	**1.11±0.17**	225±509	1.06±0.10	113±283	**1.07±0.05**	24±18
G3-N_D	M2	58	4.34±1.61	**41±6**	2.10±0.91	**54±8**	1.75±0.35	**54±4**	**1.46±0.33**	**57±3**	1.27±0.18	**60±2**
	M3	58	**5.35±1.73**	138±109	**2.36±1.30**	746±784	**1.92±0.84**	448±532	1.41±0.39	171±122	**1.31±0.20**	207±374
G5-N_D	M2	82	3.57±0.99	**55±9**	**1.72±0.66**	**74±12**	**1.50±0.39**	**74±7**	1.10±0.16	**80±5**	1.02±0.06	**82±4**
	M3	82	**5.16±1.80**	467±511	1.54±1.24	167±225	1.24±0.31	475±616	1.10±0.26	141±168	**1.04±0.09**	274±477
G7-N_D	M2	86	**2.98±0.73**	**66±13**	**1.70±0.30**	**88±9**	**1.42±0.14**	**94±4**	**1.29±0.20**	**97±4**	**1.17±0.13**	**99±2**
	M3	86	2.81±0.98	363±255	1.62±0.77	269±334	1.16±0.30	189±272	1.11±0.19	296±465	1.10±0.11	298±315

V. CONCLUSIONS

We presented an embedded software synthesis approach based on a generalized clustering approach for static dataflow subgraphs. The proposed clustering algorithm computes a quasi-static schedule and reduces the scheduling overhead for one processor of an MPSoC without an explicit state space enumeration. As a consequence, more complex systems can be handled and more compact schedules can be generated as by previously proposed approaches. Especially when considering hard memory constraints, our rule-based approach significantly outperforms state-of-the-art approaches and, thus, extends the area of applicability of such kind of embedded software synthesis. Future work will focus on optimal clustering techniques, i.e., to identify which actors should be clustered in order to minimize the scheduling overhead.

REFERENCES

[1] E. A. Lee, "The problem with threads," *Computer*, vol. 39, pp. 33–42, 2006.

[2] E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, Sep. 1987.

[3] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Cyclo-Static Dataflow," *IEEE Transaction on Signal Processing*, vol. 44, no. 2, pp. 397–408, Feb. 1996.

[4] W. Haid, K. Huang, I. Bacivarov, and L. Thiele, "Multiprocessor SoC software design flows," *Signal Processing Magazine, IEEE*, vol. 26, no. 6, pp. 64 –71, 2009.

[5] M. Thompson, H. Nikolov, T. Stefanov, A. Pimentel, C. Erbas, S. Polstra, and E. Deprettere, "A Framework for Rapid System-level Exploration, Synthesis, and Programming of Multimedia MP-SoCs," in *Proceedings of CODES-ISSS'07*, 2007, pp. 9–14.

[6] C. Haubelt, J. Falk, J. Keinert, T. Schlichter, M. Streubühr, A. Deyhle, A. Hadert, and J. Teich, "A SystemC-based Design Methodology for Digital Signal Processing Systems," *EURASIP Journal on Embedded Systems, Special Issue on Embedded Digital Signal Processing Systems*, vol. 2007, p. Article ID 47580, 2007.

[7] S. Ha, S. Kim, C. Lee, Y. Yi, S. Kwon, and Y.-P. Joo, "PeaCE: A Hardware-Software Codesign Environment of Multimedia Embedded Systems," *ACM Trans. Design Automation of Electronic Systems*, vol. 12, no. 3, pp. 1–25, 2007.

[8] S. Kwon, Y. Kim, W.-C. Jeun, S. Ha, and Y. Paek, "A Retargetable Parallel Programming Framework for MPSoC," *ACM Trans. Design Automation of Electronic Systems*, vol. 13, no. 3, 2008.

[9] L. Thiele, I. Bacivarov, W. Haid, and K. Huang, "Mapping applications to tiled multiprocessor embedded systems," in *ACSD '07: Proceedings of the Seventh International Conference on Application of Concurrency to System Design*, 2007, pp. 29–40.

[10] S. S. Bhattacharyya, J. T. Buck, S. Ha, and E. A. Lee, "Generating Compact Code from Dataflow Specifications of Multirate Signal Processing Algorithms," *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 42, no. 3, pp. 138–150, Mar. 1995.

[11] C. Hsu and S. S. Bhattacharyya, "Cycle-Breaking Techniques for Scheduling Synchronous Dataflow Graphs," Institute for Advanced Computer Studies, University of Maryland at College Park, Tech. Rep. UMIACS-TR-2007-12, Feb. 2007.

[12] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Transactions on Computers*, vol. C-36, no. 1, pp. 24–35, Jan. 1987.

[13] S. S. Bhattacharyya and E. A. Lee, "Scheduling synchronous dataflow graphs for efficient looping," *J. VLSI Signal Process. Syst.*, vol. 6, no. 3, pp. 271–288, 1993.

[14] S. S. Bhattacharyya, P. Murthy, and E. Lee, "APGAN and RPMC: Complementary Heuristics for Translating DSP Block Diagrams into Efficient Software Implementations," *Journal of Design Automation for Embedded Systems*, Jan. 1997.

[15] S. Meijer, H. Nikolov, and T. Stefanov, "Throughput modeling to evaluate process merging transformations in polyhedral process networks," in *DATE*. IEEE, 2010, pp. 747–752.

[16] W. Plishker, N. Sane, and S. Bhattacharyya, "A generalized scheduling approach for dynamic dataflow applications," in *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, 20-24 Apr. 2009, pp. 111–116.

[17] ——, "Mode grouping for more effective generalized scheduling of dynamic dataflow applications," in *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, 26-31 Jul. 2009, pp. 923–926.

[18] J. Falk, J. Keinert, C. Haubelt, J. Teich, and S. Bhattacharyya, "A Generalized Static Data Flow Clustering Algorithm for MPSoC Scheduling of Multimedia Applications," in *EMSOFT'08: Proceedings of the 8th ACM international conference on Embedded software*, Oct. 2008.

[19] S. Stuijk, M. Geilen, and T. Basten, "SDF^3: SDF For Free," in *Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings*. IEEE Computer Society Press, Los Alamitos, CA, USA, June 2006, pp. 276–278. [Online]. Available: http://www.es.ele.tue.nl/sdf3