A Generalized Static Data Flow Clustering Algorithm for MPSoC Scheduling of Multimedia Applications

Joachim Falk, Joachim Keinert, Christian Haubelt, Jürgen Teich

Hardware/Software Co-Design
Department of Computer Science
University of Erlangen-Nuremberg
Am Weichselgarten 3
91058 Erlangen, Germany

{falk,keinert,haubelt,teich}@cs.fau.de

Shuvra S. Bhattacharyya

Department of Electrical and Computer Engineering
2311 A. V. Williams Bldg.
University of Maryland
College Park MD, 20742

ssb@umd.edu

ABSTRACT

In this paper, we propose a generalized clustering approach for *static data flow subgraphs* mapped onto individual processors in Multi-Processor System on Chips (MPSoCs). The goal of clustering is to replace the static data flow subgraph by a single dynamic data flow actor such that the global performance in terms of latency and throughput is optimized. Through our proposed clustering approach, the scheduling of connected static data flow subgraphs can be coordinated with enclosing system representations in a way that systematically exploits the predictability and efficiency of the static data flow model. Thus, the advantages of static data flow subsystems can be exploited in the context of overall system representations that are based on more general models of computation. At the same time, our approach goes significantly beyond previous approaches to synchronous data flow clustering by providing a *quasi-static* — as opposed to purely-static — *scheduling interface* between clustered subgraphs and the enclosing systems. This greatly enhances the power of our techniques in terms of avoiding deadlock, increasing the design space for clustering, and providing for integration with more general models of computation. We show benefits of up to 95% performance improvement for real world examples.

Categories and Subject Descriptors

D.1.3 [**Software**]: Programming Techniques—*Concurrent Programming - Parallel Programming*

General Terms

Algorithms

Keywords

MPSoC scheduling, software synthesis, actor-oriented design

1. INTRODUCTION

Multi-Processor System on Chips (MPSoCs) are becoming more and more important as implementation platform for computationally intensive applications, e.g., multimedia or networking. As MPSoCs are mainly designed for a given set of applications, their advantage lies in a low power dissipation due to a lower clock frequency but still high computational power by exploiting the parallelism provided by the architecture. However, the high degree of parallelism of processors makes programming of MPSoCs a challenging task. While many research groups have conducted research on optimally mapping an application onto a given MPSoC platform, e.g., Daedalus [19], SystemCoDesigner [9], Koski [13], System-on-Chip Environment (SCE) [1], etc., there is still only little work on defining the right programming model for MPSoCs.

At Electronic System Level (ESL), a trend towards *actor-oriented programming models* [11, 8] can be identified. These models are often domain specific and typically used already in the mapping step. For instance, data flow models are well suited to model multimedia applications. Beside the application mapping, further refinements are needed to come from a high level model to an implementation. In particular, scheduling actors bound to one processor is one of the key issues. Often, a straight forward scheduling idea is to postpone any scheduling decision to runtime. Such a *dynamic scheduling* often results in a significant scheduling overhead and, hence, a reduced system performance. On the other hand, *static* or *quasi-static scheduling* for multi-processor systems is only solved for models with limited expressiveness, e.g., static data flow graphs. For instance, efficient single processor scheduling algorithms [2, 10] exist for *synchronous data flow models* [15], a widely accepted static data flow model.

Unfortunately, these algorithms are constrained to pure synchronous data flow graphs while real world multimedia application require modeling via heterogeneous data flow graphs also containing actors with higher levels of complex-

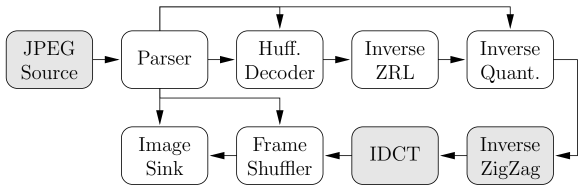

Figure 1: Example of a Motion-JPEG decoder including dynamic as well as static data flow actors (shaded). Communication between actors (vertices) is realized via FIFOs (directed edges).

ity. As an example, consider the top level data flow model of a Motion-JPEG decoder depicted in Figure 1. Beside static data flow actors (shaded vertices), i.e., actors with constant consumption or production rates known from *synchronous data flow* (SDF) models or *cyclo-static data flow* (CSDF) [5] actors, it also includes *dynamic data flow* (DDF) actors like the `Parser` which is modeled by a *Kahn process* [12]. The 2-dimensional *inverse discrete cosine transform* (`IDCT`) actor shown in Figure 1 represents the static data flow subgraph depicted in Figure 2, which consists of SDF and CSDF actors only.

After mapping the Motion-JPEG application to an MPSoC, all actors bound onto one processor have to be scheduled either dynamically or statically. As the model includes dynamic actors, a static schedule is not possible in all cases. However, a dynamic schedule may degrade the performance by introducing scheduling overheads even in the schedules of static data flow subgraphs. To permit the generation of an efficient schedule, a remedy could be the replacement of the static data flow subgraph by a single actor, i.e, clustering all static data flow actors into a new actor. Unfortunately, existing algorithms might result in infeasible schedules or greatly restrict the clustering design space by considering only SDF subgraphs that can be clustered into monolithic SDF actors without introducing deadlock. Moreover, all these approaches are limited to integrate the resulting monolithic SDF actor only with an enclosing static data flow system representation.

In this paper, we will propose a novel clustering approach for static data flow subgraphs connected to *dynamic data flow graphs*. In contrast to prior work the actor which replaces the subgraph is not restricted to SDF semantics which only allow expressing of static schedules for the actors contained in the replaced subgraph. Instead a more general actor can be generated which can have dynamic behavior. This allows expressing of a *quasi-static schedule* (QSS) for the static data flow subgraph in order to avoid deadlocks which might occur when restricting to static schedules only. The QSS is automatically derived by our clustering approach and expressed in form of a finite state machine (FSM). The created data flow actor executing this FSM can hence be easily integrated into a dynamic schedule of all remaining actors mapped onto the same processor while reducing the overall scheduling overhead. Thus, the advantages of static data flow subgraphs can be exploited by our approach in the context of more general models of computation and we improve previous work by means of increasing the design space for clustering. We show the benefits of our scheduling using a scalable benchmark as well as a real-world example, the 2-dimensional IDCT of a Motion-JPEG decoder mapped onto a single processor systems and a 4-processor system. In this case study, we could improve throughput by about 93% and 45%, respectively.

The remainder of this paper is organized as follows: Section 2 gives a motivating example. Section 3 formally defines the problem this paper is dedicated to. While Section 4 presents the proposed algorithm used by the clustering approach, we discuss related work in Section 5 and experimental results are presented in Section 6.

2. MOTIVATION AND BACKGROUND

In the following we will examine the hierarchical `IDCT` actor from Figure 1 which is depicted in its expanded version in Figure 2. As mentioned above this data flow subgraph contains only SDF (lightly shaded vertices) and CSDF (darker shaded vertices) actors. The subgraph implementing the 2-dimensional IDCT functionality is composed of two 1-dimensional IDCTs modeled by the SDF actors which are separated by the CSDF transpose actor. We will use this example to show the negative impact of *dynamic vs. static scheduling* and to reiterate important definitions for SDF and CSDF actors from [15] and [5], respectively. A *data flow schedule* is an approach to execute actors in a data flow graph such that actors *fire* only when they have sufficient data based on the underlying data flow graph model. In more detail, for each *firing* invocation an actor consumes tokens from the FIFOs connected to its input ports, executes its data processing work performed by a function in the following called *action*, and produces tokens on the FIFOs connected to its output ports. The number of tokens consumed on an input port during an actor firing is called the *consumption rate* of this input port for this firing invocation. Likewise, the number of tokens produced on an output port is called the *production rate* of this output port in this firing invocation. An actor may only fire if all the necessary tokens which will be consumed by the next actor firing are available on its input FIFOs.

To get a rough estimate of the benefits for a static system we benchmark only the IDCT part of the Motion-JPEG decoder on an embedded MicroBlaze processor running at 66MHz implemented on a Xilinx Virtex-II Pro FPGA. Then 200 2-dimensional IDCT operations on 8×8-blocks take 1.91 seconds to complete for the *dynamically scheduled* IDCT. On the other hand, using a *static schedule* for the IDCT implementation we measure a runtime of 0.95 seconds for processing the 200 blocks.

A static schedule is a data flow schedule in which the order of actor execution is determined entirely before execution begins. In contrast to this, *dynamic scheduling* wholly determines the actor execution order at runtime. Thus, due to the reduction of the overhead occurring for dynamic scheduling, processing time can be reduced by 50%. This clearly illustrates the potential of scheduling data flow actors at compile time if possible.

To exemplify this, we consider actor a_3 from Figure 2. Knowledge of the consumption rates of the actor a_3 of eight tokens per input FIFO, as annotated on the arrowheads of the edges, and the production rates of one token per output

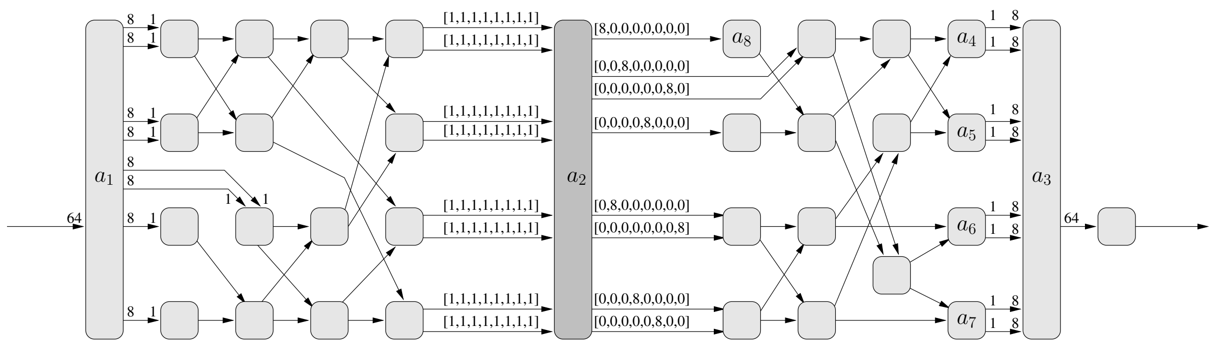

Figure 2: The above figure depicts the static data flow subgraph contained in the hierarchical IDCT actor from Figure 1. As is customary, the constant consumption and production rates of these SDF and CSDF actors are annotated on the edges, i.e., a number annotated on the arrowhead of an edge corresponds to the consumption rate of the connected actor and a number annotated on the shaft corresponds to production rate of the connected actor. Missing number annotations denote a consumption or production of one token.

FIFO of the actors a_4 to a_7, likewise annotated on the stems of the edges leaving the actor, enables us to conclude that each actor a_4 to a_7 must be fired eight times before firing actor a_3.

However, for dynamic systems, i.e, systems containing dynamic actors, no such a priori information about consumption and production rates are available. Consequently dynamic scheduling strategies, which are fitted to these systems, cannot make any assumptions about the number of produced or consumed tokens by an actor firing. Hence, no prediction is possible about actors subsequently enabled by an actor firing. Therefore, after each invocation of actors a_4 to a_7, also the precondition for actor a_3 has to be checked, i.e., whether sufficient tokens on the input FIFOs are available. However most of the times this will not be fulfilled leading to eight superfluous checks, of which seven fail and one finally succeeds but which could all have been avoided if only actor a_4 to a_7 had each been fired eight times before firing actor a_3. This eight superfluous checks are part of the scheduling overhead imposed by dynamic scheduling of static actors of the hierarchical IDCT actor. This leads to useless operations resulting in a runtime schedule overhead which is missing from a static schedule.

Formally, a *static schedule* is a sequence of actors which can be fired in the given sequence without requiring intermediate checks for preconditions, e.g., the static schedule $(8a_4, 8a_5, 8a_6, 8a_7, a_3)$ for the previously discussed actors a_3 to a_7 which denotes to fire each actor a_4 to a_7 eight times followed by one firing for a_3. At the start of such a static scheduling sequence a precondition can be checked which blocks until the execution of the whole sequence of actors can be guaranteed, e.g., checking for availability of sufficient tokens on the input ports of the subgraph defined by the actors which shall be scheduled statically.

Unfortunately static scheduling is only possible for actors which provide constant or even cyclic consumption and production rates. More formally, SDF actors are constraint to have constant consumption and production rates per port independent from the current firing invocation, i.e., for each firing invocation they produce or consume the same number of tokens on a port. Likewise, CSDF actors are constraint to have cyclic consumption and production rates per port which repeat with the same period, e.g., the CSDF actor a_2 from Figure 2 has a period of eight after which production and consumption rates repeat. To exemplify this, we consider the FIFO connected from actor a_2 to actor a_8. On the first invocation of a_2 the actor produces eight tokens on this FIFO. For the next seven invocations zero tokens are produced. At the eighth invocation the sequence repeats with the production of eight tokens.

3. PROBLEM DEFINITION

In this section, we formally define the problem this paper is dedicated to and introduce the necessary mathematical notations. The application is modeled by a so called *data flow graph*, e.g, the Motion-JPEG decoder in Figure 1 is a data flow graph:

Definition 3.1 (Data Flow Graph) *A* data flow graph *is a directed graph $g = (A, C, L)$ containing a set of* actors A *(vertices) and a set of* channels $C \subseteq A \times A$ *represented by the edges of the graph. Additionally, the data flow graph contains an initial fill level function $L : C \to \mathbb{N}_0$ which associates with each channel $(a_{\text{src}}, a_{\text{dest}}) \in C$ its number of initial tokens.*[1]

Furthermore, to ease visual references, we introduce a notion of so called *actor input ports $i \in I$* and so called *actor output ports $o \in O$* where $a.I \subseteq I$ and $a.O \subseteq O$ are the set of input and output ports of the actor a, respectively.[2] Analogously, we define the set of *subgraph input ports $g_\gamma.I$* and *subgraph output ports $g_\gamma.O$*, where $g_\gamma.I$ and $g_\gamma.O$ are actor input and output ports connected to channels crossing the subgraph boundary. An example of such an annotated data flow graph is depicted in Figure 3 where i_1, i_2 and o_1, o_2 are the input and output ports of the subgraph $g_{1,\gamma}$, respectively. Note that the proposed clustering approach is

[1]$\mathbb{N}_0 = \{0, 1, 2, 3, \ldots\}$ denotes the set of non-negative integers.

[2]We use the '.'-operator, e.g., $a.I$, for member access of tuples whose members have been explicitly named in their definition, e.g., member I of actor a.

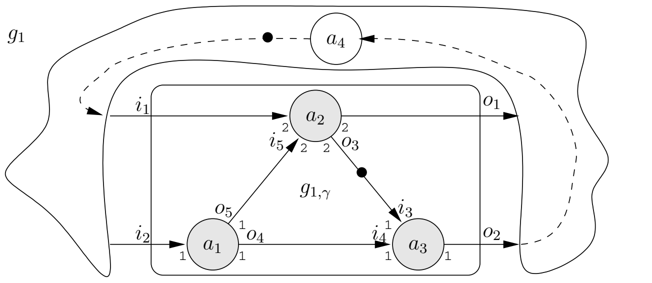

Figure 3: Example of a dynamic data flow graph g_1 with an SDF subgraph $g_{1,\gamma}$ and a maximal tight feedback loop between the subgraph output and input ports o_2 and i_1.

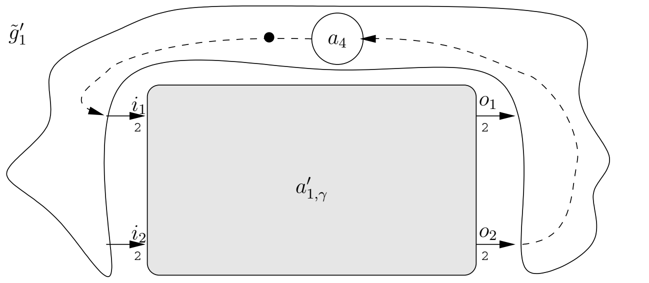

Figure 4: Converting the subgraph $g_{1,\gamma}$ from Figure 3 into an SDF actor may introduce a deadlock.

also able to consider CSDF actors. However, for the sake of readability, the examples in this paper use static data flow subgraphs containing only SDF actors.

By mapping an application modeled by a data flow graph $g = (A, C, L)$ onto an MPSoC, subgraphs of the data flow graph are bound to the distinguished programmable processors. Communications between the subgraphs are bound to the on chip buses or networks on chip. The proposed clustering approach computes for a static data flow subgraph g_γ induced by the set of static data flow actors $A_S \subseteq g.A$ bound onto a single processor of an MPSoC a *composite actor* a_γ which replaces this subgraph g_γ. This composite actor implements a data flow schedule which can vary from static over quasi-static to dynamic.

To exemplify the replacement of the subgraph by a single composite actor, we again consider Figure 3. The set of static data flow actors A_S contains the actors a_1 - a_3. Furthermore, we map the actors a_1 to a_3 and some unspecified dynamic actors of g_1 to a CPU of an MPSoC resulting in an induced subgraph $g_{1,\gamma}$ of static data flow actors mapped to this CPU. For this subgraph we have to find a composite actor to replace it. One possibility is depicted in Figure 4 where the subgraph is replaced by the SDF composite actor $a'_{1,\gamma}$ which implements the static schedule $(2a_1, a_2, 2a_3)$, i.e., fire actor a_1 twice then a_2 and finally a_3 twice. The number of firings for each actor are defined by the so called *repetition vector* which can be calculated by the *balance equation* [15]. An SDF graph is called *consistent* if the balance equation has a non-trivial solution.[3] The repetition vector assures that after execution of the static schedule sequence the graph returns into its initial state. In other words the number of tokens stored on each edge connecting the actors of the subgraph are identical before and after execution of the actor sequence. For CSDF graphs all actors furthermore have to execute a multiple of their firing period. More formally, we define clustering as follows:

Definition 3.2 (Clustering) *Given a data flow graph g and a static data flow subgraph g_γ induced by all static data flow actors $a \in A_S \subseteq g.A$ which are bound onto a given processor of the considered MPSoC. Clustering replaces g_γ by a single actor a_γ, called* composite actor*, implementing a data flow schedule for the actors $a \in A_S$, resulting in a new data flow graph $\tilde{g}$, i.e., $\tilde{g}.A = g.A + \{a_\gamma\} - A_S$ and $\tilde{g}.C = g.C + C_a - C_S$, where $C_S = \{c = (a_1, a_2) \in g.C \mid a_1 \in A_S \vee a_2 \in A_S\}$ and C_a is the set of edges connecting a_γ with the remaining DDF actors $g.A - A_S$, i.e., $C_a = \{(a_{src}, a_{sink}) \in \tilde{g}.A \times \tilde{g}.A \mid (a_{src} = a_\gamma \implies \exists (a', a_{sink}) \in g.C : a' \in A_S) \vee (a_{sink} = a_\gamma \implies \exists (a_{src}, a') \in g.C : a' \in A_S)\}$.*

[3]The trivial solution being the all zeros repetition vector.

The goal of our generalized methodology for data flow graph clustering is to allow designers and design tools more flexibility in trading off among different costs and benefits related to the grouping of subgraphs into individual units for scheduling. The specific clustering algorithm that we develop in this paper is geared towards reducing dynamic scheduling overhead, while providing enough flexibility to avoid deadlock during the clustering process, i.e., the data flow schedule implemented by the composite actor is as static as possible. A static schedule can be considered to have obtained the full benefits of the given information contained in the static subgraph, whereas a quasi static schedule represents a intermediate level of success and a dynamic schedule is the degenerated case. The clustering algorithm produces a quasi-static schedule in the sense that any scheduling decisions which depend on preconditions which cannot be predicted at compile time are still made at runtime. Actor firings which can be executed as consequence of a precondition are scheduled at compile time resulting in a static schedule for this actor firings. The static and dynamic cases for the data flow schedule can be considered the best case and the degenerated cases of the quasi-static schedule, respectively.

Exploration of clustering techniques that exploit the flexibility of our generalized framework for other scheduling objectives is a useful direction for further work. The more restricted approach of replacing a static data flow subgraph by its corresponding SDF actor implementing a static schedule for the actors of the subgraph excels at schedule overhead reduction.[4] An example of such a best case scenario is shown in Figure 2 where the SDF subgraph for the 2-dimensional `IDCT` can be replaced by an SDF composite actor implementing a static schedule of the contained IDCT actors, i.e., for the composite actor replacing the `IDCT` subgraph we only need to check the prerequisite of 64 tokens on the input of the `IDCT` before we can fire the contained static schedule.

However, this approach might be infeasible due to the environment, e.g., as depicted in Figure 3, of the static data

[4]In general the longer the static scheduling sequence which can be fired by checking a single prerequisite, the less schedule overhead is imposed by this schedule.

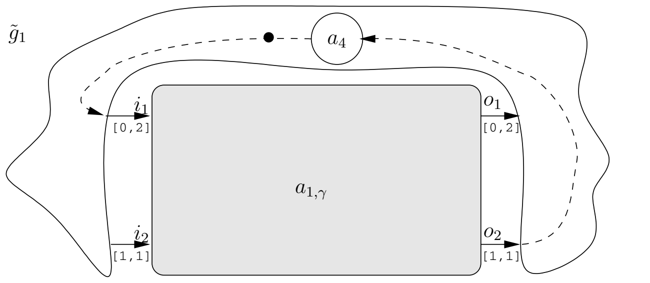

Figure 5: Instead of a conversion of the subgraph $g_{1,\gamma}$ into an SDF an alternative conversion into the above depicted CSDF actor solves the introduction of the deadlock from Figure 4.

flow subgraph $g_{1,\gamma}$ displayed as a cloud in all figures. The problem stems from the lack of flexibility in the static schedule $(2a_1, a_2, 2a_3)$ implemented by the composite actor $a'_{1,\gamma}$ (cf. Figure 4) and the feedback loop between the output o_2 to the input i_1 (represented by the dashed edge and the dynamic data flow (DDF) actor a_4) that imposes the additional constraint on the schedule that actor a_3 has to be executed at least once before executing actor a_2 in order to avoid a deadlock. A constraint which is violated by the static schedule $(2a_1, a_2, 2a_3)$. This situation can be solved by converting the subgraph $g_{1,\gamma}$ into a CSDF composite actor $a_{1,\gamma}$ as depicted in Figure 5. This actor $a_{1,\gamma}$ implements a quasi-static schedule composed of two static schedules. First a check for one token on port i_2 is performed, then the static schedule (a_1, a_3) is fired. Next a check follows for two tokens on port i_1 and one token on port i_2, then the static schedule (a_1, a_2, a_3) is executed.

However, this strategy fails when applied to the next example depicted in Figure 6. In this case we assume a dynamic actor a_4 which uses two different forward modes. In mode (i), it forwards both depicted tokens to port i_1. This allows production of two tokens on port o_1 which are then forwarded to port i_2 by actor a_4. In mode (ii) however, actor a_4 starts to forward one token to port i_2. This time the subgraph $g_{1,\gamma}$ can generate one token at output o_2 which allows actor a_4 to forward two tokens to port i_1. After generation of two tokens on output port o_1, actor a_4 finally forwards again one token to port i_2.

Unfortunately we can recognize that applying the above schedule, (a_1, a_3) followed by (a_1, a_2, a_3), fails if actor a_4 forwards according to forward mode (i). We could try to solve this problem by substituting another CSDF composite actor $a'_{2,\gamma}$ (cf. Figure 7) that implements the check for two tokens on i_1, fires schedule (a_2), checks for two tokens on i_2 and finally fires schedule $(2a_1, 2a_3)$. However, this schedule fails for forward mode (ii).

More explicitly the forward mode (i) of actor a_4 requires firing actor a_2 first while forward mode (ii) imply firing (a_1, a_3) first. As the dynamic actor can arbitrarily switch between the two modes, the subgraph $g_{2,\gamma}$ can neither be converted into an SDF nor a CSDF actor without possibly introducing a deadlock. Therefore, the decision which schedule sequence to choose depends on token availability on the subgraph input ports i_1 and i_2. However, this flexibility cannot be implemented by a static schedule. Instead a Quasi-Static Schedule (QSS) is required which statically schedules the sequences (a_1, a_3) and a_2 but which postpones the decision of the sequence to execute first to runtime.

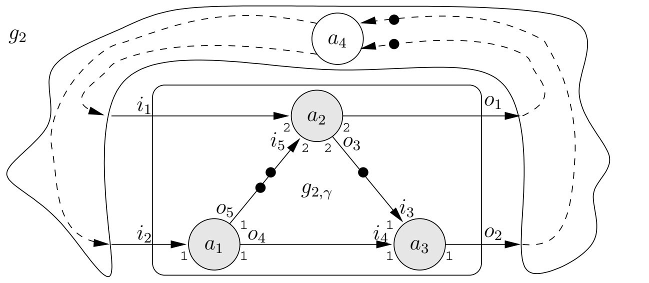

Figure 6: Second example with two feedback loops $o_1 \rightarrow i_2$ and $o_2 \rightarrow i_1$ over the dynamic data flow actor a_4 which may randomly switch between the forwarding modes (i) two tokens forwarded to port i_1 followed by two tokens to port i_2 and (ii) one token to port i_2, two tokens to port i_1 followed by one token to port i_2.

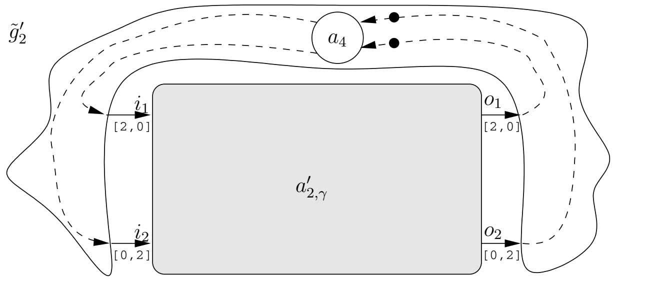

Figure 7: Second try for a CSDF composite actor replacing the subgraph $g_{2,\gamma}$.

The above examples demonstrate that in the general case a static data flow subgraph cannot be converted into an SDF actor nor a CSDF actor without introducing a possible deadlock into the transformed data flow graph. In the following, we introduce the so called *cluster Finite State Machine (FSM)* representation for composite actors providing this flexibility. The cluster FSM is an immediate representation for the quasi-static schedules computed by our clustering algorithm. More formally, we define:

Definition 3.3 (Cluster FSM) *The* cluster FSM *of a composite actor a_γ is a tuple $m = (Q, q_0, T, N, R)$ containing a finite set of* states Q *and an* initial state $q_0 \in Q$*, a finite set of* transitions $(q_{\text{src}}, q_{\text{dest}}) \in T \subseteq Q \times Q$*, a* guard function $N : T \rightarrow \mathbb{N}_0{}^{|g_\gamma.I|}$ *specifying the precondition for the number of tokens required on each channel connected to the subgraph input ports $g_\gamma.I$ to execute a transition, and finally an* action function $R : T \rightarrow g_\gamma.A^*$ *encoding a static scheduling sequence for the actors $a \in g_\gamma.A$ of the subgraph.*

With this notation, the subgraph $g_{1,\gamma}$ can be replaced by a composite actor $a_{2,\gamma}$ which can be expressed as depicted in Figure 8. Note that we use $\#i$ to denote the number of available tokens on the channel connected to the actor input port

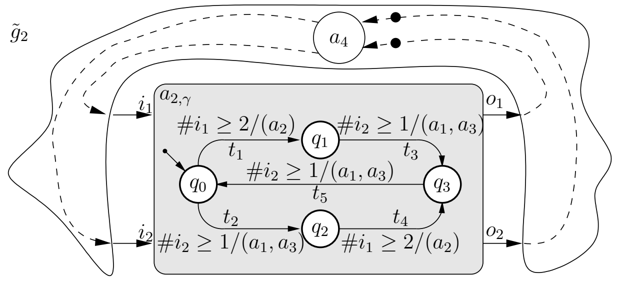

Figure 8: The composite actor $a_{2,\gamma}$ replacing subgraph $g_{2,\gamma}$ represented via a *cluster Finite State Machine.* The transitions are annotated with preconditions and the static scheduling sequences to execute if a transition is taken.

i. As it can be seen, two transitions t_1 and t_2 are leaving the start state q_0. t_1 requires at least two tokens on input port i_1 denoted by the precondition $\#i_1 \geq 2$ and executes the static schedule (a_2) whereas t_2 requires at least one input token on input port i_2 denoted by $\#i_2 \geq 1$ and executes the static schedule (a_1, a_3). This notation allows us to represent static schedules (The FSM has one state with one transition self-loop), quasi-static schedules (The FSM has at least one transition with a scheduling sequence of more then one actor), and dynamic schedules (All transitions of the FSM have a degenerated scheduling sequence containing exactly one actor).

4. CLUSTERING ALGORITHM

In this section, we will present a methodical way to construct the cluster FSM m as defined in Definition 3.3 that represents a quasi-static schedule (QSS) for a given static data flow subgraph g_γ. The key idea to our algorithm is that each output $o \in g_\gamma.O$ of g_γ might have a feedback via other data flow actors to each input $i \in g_\gamma.I$. As we have learned from the examples in Section 3, any produced token on an output $o \in g_\gamma.O$ might cause through these feedback loops the activation of an actor $a \in g_\gamma.A$ in the same subgraph. In particular, postponing the production of an output token may result in a deadlock of the entire system. Hence, the QSS determined by our clustering algorithm guarantees the production of a maximum number of output tokens from the consumption of a minimal number of input tokens. However, the proposed clustering algorithm requires that tokens produced by an output port depend on all input ports. Otherwise an unbounded accumulation of tokens on a channel inside the cluster may result. Therefore, resulting in an infinite state space of the cluster FSM, due to the subsumption of the cluster channel fill levels in the state of the cluster FSM. More formally, we define the following clustering condition:

Definition 4.1 (Clustering Condition) *A static data flow subgraph g_γ can be clustered by the given algorithm if the subgraph disregarding its inputs and outputs is deadlock free itself and for each pair of actors $(a_{\text{src}}, a_{\text{dest}})$ possessing a subgraph input and output port there exists a directed path $p \in g_\gamma.C^*$ from actor a_{src} to actor a_{dest}, i.e., $\forall a_{\text{src}}, a_{\text{dest}} \in g_\gamma.A, a_{\text{src}} \neq a_{\text{dest}} : (a_{\text{src}}.I \cap g_\gamma.I \neq \emptyset \wedge a_{\text{dest}}.O \cap g_\gamma.O \neq \emptyset) \implies \exists$ a directed path $p = ((a_{\text{src}}, a_2), (a_2, a_3), \ldots, (a_{n-1}, a_{\text{dest}})) \in g_\gamma.C^*$.*

To exemplify this consider the subgraph $g_{2,\gamma}$ depicted in Figure 6 satisfying the clustering condition. However, removing the channel (a_1, a_2) would contradict the condition as there is no path from i_2 to o_1 and introduce an unbounded accumulation of tokens on edge (a_2, a_3) while producing tokens on port o_1 but never requiring any tokens on i_2. For static data flow subgraphs that do not satisfy the cluster condition, the set of static actors A_{S} can be partitioned into subsets each inducing a connected subgraph satisfying the *clustering condition.*

Our proposed clustering algorithm works in three steps: (**step 1**) Preprocessing, (**step 2**) Compute the set of *input/output states* each representing the maximal production of output tokens with a minimal consumption of input tokens, and (**step 3**) Construct the cluster FSM. In the following, we discuss these individual steps in detail.

The preprocessing computes some termination criteria for **step 2** and **step 3**. In particular, the number of firings of each actor to bring the cluster back into its initial state as well as the number of consumed and produced tokens by these firings will be computed. More formally:

Step 1.1: Compute the *repetition vector* [15] $\mathbf{r}_{\min,g_\gamma}$ for subgraph g_γ, i.e., a positive integer $\mathbf{r}_{\min,g_\gamma}(a)$ is assigned to each actor $a \in g_\gamma.A$ in the subgraph denoting the minimal number of firings of a to return g_γ back to its initial state. For the subgraph $g_{2,\gamma}$ in Figure 6, the repetition vector is $\mathbf{r}_{\min,g_{2,\gamma}} = (n_{a_1}, n_{a_2}, n_{a_3}) = (2, 1, 2)$, i.e., actors a_1 and a_3 have to be fired twice, whereas actor a_2 has to be fired once.

Step 1.2: Compute the so called *input/output repetition vector* $\mathbf{n}_{\min,g_\gamma}$ that assigns to each input $i \in g_\gamma.I$ and each output $o \in g_\gamma.O$ the number of consumed tokens $\mathbf{n}_{\min,g_\gamma}(i)$ or produced token $\mathbf{n}_{\min,g_\gamma}(o)$, respectively, by firing each actor $a \in g_\gamma.A$ exactly $\mathbf{r}_{\min,g_\gamma}(a)$ times. For example, the input/output repetition vector of $g_{2,\gamma}$ computes to $\mathbf{n}_{\min,g_{2,\gamma}} = (n_{i_1}, n_{i_2}, n_{o_1}, n_{o_2}) = (2, 2, 2, 2)$, i.e., from each input, two tokens are consumed and on each output, two tokens are produced when firing actors a_1 and a_3 twice and actor a_2 once.

In order to avoid deadlocks, we assume the worst case, i.e., each produced output token causes the activation of an actor in the subgraph via a feedback loop. Hence, it is required that the resulting QSS always produces a maximal number of output tokens with a minimal number of input tokens. Each end point of such a production is marked by an *input/output state* of the subgraph. As an exhaustive evaluation is prohibitive, we propose the following three steps to determine the input/output states:

Step 2.1: Compute for each output port o the *input/output dependency* tuples encoding the minimal numbers of consumed tokens on the input ports to produce n tokens on o. For this purpose, we formally define an *input/output dependency function* dep_{g_γ}.

Definition 4.2 (Input/Output Dependency Function) *Given a subgraph g_γ the* input/output dependency function $\text{dep}_{g_\gamma} : g_\gamma.O \times \mathbb{N}_0 \to \mathbb{N}_0{}^{|g_\gamma.I|}$ *is a function that associates with a cluster g_γ, for each subgraph output port $o \in g_\gamma.O$,* *and for a requested number of tokens $n \in \mathbb{N}_0$ a vector of minimal number of input tokens $(n_{i_1}, n_{i_2}, \ldots n_{i_{|g_\gamma.I|}})$ consumed on each subgraph input port $i \in g_\gamma.I$ to produce the requested number n of tokens on the output port o.*

Note that the set of input/output dependency values we need to consider can be bounded by the input/output repetition vector, i.e., we only need to consider the set $\mathrm{iodep}_{g_\gamma} = \{\mathrm{dep}_{g_\gamma}(o,n) \mid o \in g_\gamma.O, n \in \mathbb{N}_0, \mathrm{dep}_{g_\gamma}(o,n) \not> \mathbf{m}_{\min,g_\gamma}\}$ containing tuples not greater than the projection of the input/output repetition vector $\mathbf{n}_{\min,g_\gamma} = (n_{i_1}, n_{i_2}, \ldots n_{i_{|g_\gamma.I|}}, n_{o_1}, n_{o_2}, \ldots n_{o_{|g_\gamma.O|}})$ to its inputs $\mathbf{m}_{\min,g_\gamma} = (n_{i_1}, n_{i_2}, \ldots n_{i_{|g_\gamma.I|}})$. The algorithm to calculate the input/output dependency function is depicted below, where fill(c) defines the number of tokens stored on the channel c. The $\leftarrow$ sign is used to indicate variable assignment.

Algorithm 4.1 dep

```
IN:  The cluster $g_\gamma$,
     the cluster output port $o \in g_\gamma.O$,
     the number of requested tokens $n$ on output $o$.
OUT: The minimal number of tokens required on each
     input $i \in g_\gamma.I$ to produce $n$ tokens on the output $o$.
BEGIN
  LET $\forall c \in g_\gamma.C : \mathrm{fill}(c) \leftarrow g_\gamma.L(c)$
      $\forall p \in g_\gamma.I \cup g_\gamma.O \;:\; \mathrm{fill}(p) \;\leftarrow\; \begin{cases} -n & \text{if } p = o \\ 0 & \text{otherwise} \end{cases}$
  WHILE $\exists a \in g_\gamma.A, o \in a.O : \mathrm{fill}(o) < 0$ DO [5]
    LET $n \leftarrow \max\left\{ \left\lceil \frac{-\mathrm{fill}(o)}{\mathrm{prod}(o)} \right\rceil \mid o \in a.O \right\}$
    $\forall i \in a.I : \mathrm{fill}(i) \leftarrow \mathrm{fill}(i) - n \cdot \mathrm{cons}(i)$
    $\forall o \in a.O : \mathrm{fill}(o) \leftarrow \mathrm{fill}(o) + n \cdot \mathrm{prod}(o)$
  DONE
  RETURN $-(\mathrm{fill}(i_1), \mathrm{fill}(i_2), \ldots \mathrm{fill}(i_{|g_\gamma.I|}))$
END
```

For the subgraph $g_{2,\gamma}$, the input/output dependencies are shown in Table 1.
Note that the algorithm works with negative channel fill levels and therefore cannot be used to check for the existence of a valid schedule for a given consistent SDF subgraph. These schedulability checks are instead performed in Step 3.5 where the static scheduling sequences for each transition in the cluster FSM are derived. Furthermore, a failure of the schedulability check is equivalent to the existence of a deadlock in the original subgraph g_γ.
Note that the computation of input/output dependencies is bounded by the input/output repetition vector, i.e., it is only necessary to consider values n where $\mathrm{dep}_{g_\gamma}(o,n) \not\geq \mathbf{n}_{\min,g_\gamma}$, i.e., $\mathrm{iodep}_{g_\gamma} = \{\mathrm{dep}_{g_\gamma}(o,n) \mid o \in g_\gamma.O, n \in \mathbb{N}_0, \mathrm{dep}_{g_\gamma}(o,n) \not\geq \mathbf{n}_{\min,g_\gamma}\}$.

Step 2.2: Compute the so called *input/output state set* $\mathrm{iostates}_{g_\gamma}$ by determining the maximal number of output tokens producible on each output port for the number of input tokens available in each input/output dependency value. Additionally, add the null input/output state $(0, 0, \ldots 0)$ which may be missing if the subgraph can produce output tokens without consuming any inputs. More formally, we define the input/output state set as follows:

[5]Please note that actor ports are used interchangeable with the channels connected to them.

Table 1: Input/output dependency values for subgraph $g_{2,\gamma}$ from Figure 6 and corresponding input/output states. For example, to produce at least $n = 1$ token on output $o = o_1$, at least two tokens are required from input port i_1.

	$\mathrm{dep}_{g_{2,\gamma}}(o,n)$		$\mathrm{iostates}_{g_{2,\gamma}}$	
	$o = o_1$	$o = o_2$		
$n = 0$	$(0,0)$	$(0,0)$	$(0,0,0,0)$	$(0,0,0,0)$
$n = 1$	$(2,0)$	$(0,1)$	$(2,0,2,0)$	$(0,1,0,1)$
$n = 2$	$(2,0)$	$(2,2)$	$(2,0,2,0)$	$(2,2,2,2)$
$n = 3$	$(4,2)$	$(2,3)$	—	—

Definition 4.3 (Input/Output State Set) *The input/output state set* $\mathrm{iostates}_{g_\gamma} \subseteq \mathbb{N}_0^{|g_\gamma.I \cup g_\gamma.O|}$ *consists of input/output states* $(n_{i_1}, n_{i_2}, \ldots n_{i_{|g_\gamma.I|}}, n_{o_1}, n_{o_2}, \ldots n_{o_{|g_\gamma.O|}})$ *encoding the minimal number of required input tokens* n_i *per input port* $i \in g_\gamma.I$ *and the maximal producible output tokens* n_o *per output port* $o \in g_\gamma.O$ *from this input tokens, i.e.,*

$$\begin{aligned}
\mathrm{iostates}_{g_\gamma} = \{&(0,0,\ldots 0)\} \cup \{ \\
&(n_{i_1}, n_{i_2}, \ldots n_{i_{|g_\gamma.I|}}, n_{o_1}, n_{o_2}, \ldots n_{o_{|g_\gamma.O|}}) \\
\mid\ & (n_{i_1}, n_{i_2}, \ldots n_{i_{|g_\gamma.I|}}) \in \mathrm{iodep}_{g_\gamma} \\
\wedge\ & \forall o \in g_\gamma.O : n_o = \max(\\
&\quad \{n \in \mathbb{N}_0 \mid \mathrm{dep}_{g_\gamma}(o,n) \leq (n_{i_1}, n_{i_2}, \ldots n_{i_{|g_\gamma.I|}})\}) \qquad \}.
\end{aligned}$$

For the subgraph $g_{2,\gamma}$ in Figure 6, the input/output state set $\mathrm{iostates}_{g_{2,\gamma}}$ computes to $\{(0,0,0,0), (2,0,2,0), (0,1,0,1), (2,2,2,2)\}$ (cf. Table 1). Note that the entries for $n = 3$ are not contained in $\mathrm{iodep}_{g_\gamma}$ due to their redundancy.

Step 2.3: In the previous two steps, we have neglected the different interleavings of actor firings that are permitted by the partial order of these actor firings. In Figure 6 for instance, we can fire actor a_2 followed by one execution of sequence (a_1, a_3) which results in the iostate $(2,1,2,1)$. As however algorithm 4.1 considers each output port individually, this iostate does not occur in $\mathrm{iostates}_{g_{2,\gamma}}$.

This can be solved by computing the least fixpoint $\mathrm{lfp}(\mathrm{iostates}_{g_\gamma})$ which is defined as the pointwise maximum of all pairs of input/output states $\mathbf{n} \in \mathrm{iostates}_{g_\gamma}$. For our example given in Figure 6, the least fixpoint computes to $\mathrm{lfp}(\mathrm{iostates}_{g_\gamma} = \{(0,0,0,0), (2,0,2,0), (0,1,0,1), (2,1,2,1), (2,2,2,2)\}$.
Note that the state $(2,1,2,1)$ has been added to the input/output state set.

After computing the input/output states, the cluster FSM can be constructed by ordering the input/output states and computing the transitions between input/output states. This can be done by the following five steps:
Step 3.1: Compute the partial order $\mathbf{n}_1 \leq \mathbf{n}_2$ on $\mathrm{lfp}(\mathrm{iostates}_{g_\gamma})$ where $\mathbf{n}_1 \leq \mathbf{n}_2$ iff $\forall p \in g_\gamma.I \cup g_\gamma.O : \mathbf{n}_1(p) \leq \mathbf{n}_2(p)$. We use Hasse diagrams to visualize partial orders, where vertices represent input/output states and directed edges $(\mathbf{n}_1, \mathbf{n}_2)$ represent the relation

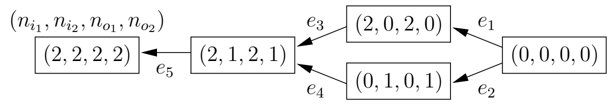

Figure 9: Hasse diagram of the partial order for the input/output states $\mathrm{lfp}(\mathrm{iostates}_{g_{2,\gamma}})$ **of the subgraph** $g_{2,\gamma}$ **shown in Figure 6.**

$\mathbf{n}_1 \leq \mathbf{n}_2$. The resulting Hasse diagram of $\mathrm{lfp}(\mathrm{iostates}_{g_{2,\gamma}})$ computed in **step 2.3** is depicted in Figure 9. This partial order implies the state transitions for the cluster FSM as can be seen later. Furthermore, we use $\mathcal{TO}_{g_\gamma}$ to denote the set of so called *tightly ordered pairs* $(\mathbf{n}_{\mathrm{src}}, \mathbf{n}_{\mathrm{dest}}) \in \mathrm{lfp}(\mathrm{iostates}_{g_\gamma})^2$ where $\mathbf{n}_{\mathrm{src}} < \mathbf{n}_{\mathrm{dest}}$ but no input/output state $\mathbf{n}'$ exists between $\mathbf{n}_{\mathrm{src}}$ and $\mathbf{n}_{\mathrm{dest}}$, i.e., $\mathcal{TO}_{g_\gamma} = \{(\mathbf{n}_{\mathrm{src}}, \mathbf{n}_{\mathrm{dest}}) \in \mathrm{lfp}(\mathrm{iostates}_{g_\gamma})^2 \mid \mathbf{n}_{\mathrm{src}} < \mathbf{n}_{\mathrm{dest}} \wedge \nexists \mathbf{n}' \in \mathrm{lfp}(\mathrm{iostates}_{g_\gamma}) : \mathbf{n}_{\mathrm{src}} < \mathbf{n}' < \mathbf{n}_{\mathrm{dest}}\}$. Hence, the *tightly ordered pairs* are the edges in the corresponding Hasse diagram, e.g., $e_3 = ((2,0,2,0),(2,1,2,1)) \in \mathcal{TO}_{g_{2,\gamma}}$ in Figure 9. Finally, we define a modulo operation $\mathbf{n} \bmod \mathbf{n}_{\min,g_\gamma}$ to be the greatest vector $\mathbf{n}' = \mathbf{n} + n \cdot \mathbf{n}_{\min,g_\gamma}$ not greater than or equal to $\mathbf{n}_{\min,g_\gamma}$, where $n \in \mathbb{Z}$. For example, $(1,3) \bmod (2,2) = (1,3)$ but $(2,3) \bmod (2,2) = (0,1)$. Later, the modulo operation is used in step 3.3 to determine the set of transitions T of the cluster FSM.

Step 3.2: Compute the state set Q of the cluster FSM m by generating a state $q \in Q$ for each input/output state $\mathbf{n}$ not greater than or equal to the input/output repetition vector, i.e., $m.Q = \{\mathbf{n} \in \mathrm{lfp}(\mathrm{iostates}_{g_\gamma}) \mid \mathbf{n} \not\geq \mathbf{n}_{\min,g_\gamma}\}$. The bottom element in the partial order, i.e., the null input/output state $(0,0,\ldots 0)$, corresponds to the initial state q_0. For the example in Figure 6, the state set computes to $Q = \{q_0 = (0,0,0,0), q_1 = (2,0,2,0), q_2 = (0,1,0,1), q_3 = (2,1,2,1)\}$ (cf. Figure 8).

Step 3.3: Compute the transition set T of the cluster FSM m by generating a transition $t \in T$ for each tightly ordered pair of input/output states, i.e., $m.T = \{(\mathbf{n}_{\mathrm{src}} \bmod \mathbf{n}_{\min,g_\gamma}, \mathbf{n}_{\mathrm{dest}} \bmod \mathbf{n}_{\min,g_\gamma}) \mid (\mathbf{n}_{\mathrm{src}}, \mathbf{n}_{\mathrm{dest}}) \in \mathcal{TO}_{g_\gamma}\}$. Considering $\mathcal{TO}_{g_{2,\gamma}} = \{e_1, e_2, e_3, e_4, e_5\}$ in Figure 9, the resulting set of transitions is $T = \{(q_0,q_1),(q_0,q_2),(q_1,q_3),(q_2,q_3),(q_3,q_0)\}$. Note that only for transition $t_5 = (q_3, q_0)$ the modulo operation was necessary.

Step 3.4: Compute the guard function N of the cluster FSM m by generating a guard function value $N(t)$ encoding the minimal number of tokens on each subgraph input port to enable the transition t from each tightly ordered pair of input/output states, i.e., $\forall (\mathbf{n}_{\mathrm{src}}, \mathbf{n}_{\mathrm{dest}}) \in \mathcal{TO}_{g_\gamma} : m.N(\mathbf{n}_{\mathrm{src}} \bmod \mathbf{n}_{\min,g_\gamma}, \mathbf{n}_{\mathrm{dest}} \bmod \mathbf{n}_{\min,g_\gamma}) = (n_{i_1}, n_{i_2}, \ldots n_{|g_\gamma.I|})$ where $n_i = \mathbf{n}_{\mathrm{dest}}(i) - \mathbf{n}_{\mathrm{src}}(i)$. Considering e_5, at least one token, i.e., $((2,2,2,2) - (2,1,2,1))(i_2) = 1$, on port i_2 is necessary to enter state q_0.

Step 3.5: Compute the action function R of the cluster FSM m. For this purpose, a *partial repetition vector* $\mathbf{r}_{g_\gamma,t}$ that assigns to each actor $a \in g_\gamma.A$ a non-negative integer $\mathbf{r}_{g_\gamma,t}(a)$ indicating the number of actor firings to go from state q_{src} to state q_{dest} is derived from each tightly ordered pair of input/output states, i.e., $\mathbf{r}_{g_\gamma,t} = \mathbf{r}_{\mathrm{dest}} - \mathbf{r}_{\mathrm{src}}$ where $t = (q_{\mathrm{src}}, q_{\mathrm{dest}})$, $q_{\mathrm{src}} = \mathbf{n}_{\mathrm{src}} \bmod \mathbf{n}_{\min,g_\gamma}$, and $q_{\mathrm{dest}} = \mathbf{n}_{\mathrm{dest}} \bmod \mathbf{n}_{\min,g_\gamma}$. Furthermore, the repetition vectors $\mathbf{r}_{\mathrm{dest}}$ and $\mathbf{r}_{\mathrm{src}}$ correspond to the input/output states $\mathbf{n}_{\mathrm{dest}}$ and $\mathbf{n}_{\mathrm{src}}$, e.g., firing each actor $a \in g_\gamma.A$ $\mathbf{r}_{\mathrm{src}}(a)$ times produces $\mathbf{n}_{\mathrm{src}}(o)$ tokens on output port o and consumes $\mathbf{n}_{\mathrm{src}}(i)$ tokens from input port i. Finally, for each transition on the basis of the partial repetition vector, a single processor schedule is computed by a version of the scheduling algorithm presented in [10] modified to support partial repetition vectors. This schedule is assigned to $R(t)$ and is one of the schedule phases of the resulting composite actor replacing the static data flow subgraph g_γ.

5. RELATED WORK

In this section, we will present the related work pertaining to clustering of synchronous data flow graphs. All the presented approaches only convert an SDF subgraph to an SDF actor. The advantages of clustering SDF actors for the purpose of generating static schedules have been shown in [3] which introduced the *Pairwise Grouping of Adjacent Nodes (PGAN)* clustering algorithm for constructing lexical orderings for later conversion into *single appearance schedules.* However, to detect the feasibility of a clustering operation, this algorithm uses the corresponding *Acyclic Precedence Graph (* APG*)* [16] of the SDF graph, a representation which can grow exponentially in the number of actors in the data flow graph. Due to the restriction that the APG can only be derived for synchronous data flow graphs, heterogeneous data flow systems cannot be clustered using this algorithm. An improvement of PGAN for acyclic graphs has been presented by Bhattacharyya et al. [4] with *Acyclic Pairwise Grouping of Adjacent Nodes (APGAN)* algorithm. Another complementary heuristic for clustering is the top-down algorithm *Recursive Partitioning by Minimum Cuts (RPMC)* also presented in this work. Both APGAN and RPMC could in principle be used for clustering heterogeneous data flow systems due to the restriction that only acyclic graphs are handled. This evades the problem of feedback loops if we also require an infinite input to the SDF subgraph in the heterogeneous graph. Otherwise only a prefix of the output stream may be generated by the resulting composite actor. In [17] a heuristic for SDF clustering is presented which avoids the exponential growth problem of PGAN but is also limited to generating SDF actors.

Additionally, clustering is used for MPSoC scheduling by clustering actors which are later bound to a dedicated resource. In [14], a technique is developed to cluster dataflow subgraphs to guide multiprocessor scheduling techniques towards solutions that provide lower schedule makespan (i.e., that minimize the time required to execute a single iteration of a dataflow graph). However, this technique is limited to operating on homogeneous SDF graphs. The technique cannot handle any kind of dynamic dataflow graph, nor can it process SDF graphs unless they are first expanded into equivalent homogeneous SDF graphs.

Scheduling SDF subgraphs within dynamic dataflow graphs to decrease scheduling overhead is explored in [6, 7]. However, in these approaches, the clustered SDF graphs are treated as atomic (pure SDF) actors, and therefore the

Table 2: Measured runtimes for different synthetic graphs and 1000 cluster FSM cycles.

#Actors	QSS [s]	dynamic [s]	improvement
12	1.2	3.0	60%
24	5.3	7.5	29%
39	10.3	14.0	26%

clustering design space and the resulting schedules are more restricted compared to those associated with the approach that we present in this paper. Finally, Vincentelli et al. [18] presented a quasi-static scheduling approach for equal conflict Petri nets. However, this technique is limited to pure equal conflict nets and exhibits exponentially complexity in the number of conflicts. If only a subgraph exhibiting equal conflict semantics is scheduled with this technique a best case environment is assumed, i.e., the feedback loop problem over the environment of the subgraph is neglected.

6. RESULTS

In order to illustrate the benefits of our clustering algorithm developed in Section 4, we have applied it to both synthetic data flow graphs as well as the IDCT of a Motion JPEG Decoder. The synthetic graphs are generated based on Figure 6 by adding a variable number of actors with identical delays on edges $(a_4, a_2), (a_2, a_4), (a_1, a_3)$.

Table 2 shows the obtained measurements dependent on the number of actors in the subgraph. It compares the runtime when applying the QSS determined by the clustering algorithm against a dynamic scheduler. The latter one polls each actor in a round-robin fashion whether it can execute or not. The achieved improvements clearly show the benefits of our clustering algorithm which is able to derive quasi-static schedules even when the cluster cannot be represented by a static data flow actor. Note that the decreasing improvement with increasing problem size is due to our particular synthetic example in combination with the implemented round-robin scheduler which accidentally schedules the static clusters nearly optimal. However, this observation cannot be generalized.

In order to evaluate the optimization potential for real-world examples, we additionally have applied our clustering algorithm to the two-dimensional IDCT of our Motion-JPEG decoder for both, a single-processor and a multi-processor implementation. Both of them were implemented using embedded MicroBlaze processors running at 66 MHz a Xilinx Virtex-II Pro FPGA. For the single-processor implementation with round-robin scheduling, we derived a latency of 1.91 seconds per 200 blocks and 0.95 seconds using the QSS computed by our proposed method. This results in a latency improvement of 50%. The throughput improvement is even more significant and amounts to 93% (round-robin scheduling: 110 blocks/s, QSS: 212blocks/s).

For the multi-processor implementation, we partitioned the IDCT into four clusters as shown in Figure 10. Each of the four composite actors resulting from applying our clustering algorithm is implemented on a single MicroBlaze. Inter-processor communication is implemented using Xilinx Fast Simplex Links (FSL). The source (s) and the sink (d) are implemented as hardware modules. Note that each of the four clusters satisfies the clustering condition from Definition 4.1.

Comparing the dynamic round-robin scheduler with the QSS resulting from the clustering algorithm, the latter improves the throughput by 45% (round-robin: 1259 blocks/s, QSS: 1831 blocks/s).[6] The latency improvement is slightly smaller with 34% (round-robin: 0.17 ms per 200 blocks, QSS: 0.11 ms per 200 blocks).

The performance increase can be explained by two facts. First of all, the dynamic round-robin scheduler polls each actor individually whether it can fire or not. Due to the unequal number of invocations however, several pollings fail, wasting precious computational power. Additionally, the dynamic scheduler has to check the fill level of each FIFO connecting two actors, even if both of them are mapped to the same processor. The QSS schedule on the other hand only checks the hardware input and output FIFOs. As soon as there are enough input tokens, a sequence of actor firings determined during compile time is executed without checking the internal FIFOs.

7. CONCLUSIONS

In this paper, we presented a generalized clustering approach for static data flow subgraphs and proposed a clustering algorithm that computes a quasi-static schedule reducing the scheduling overhead for one processor of an MPSoC while still accommodating a worst-case environment of the cluster. Through our proposed clustering approach, the scheduling of these subgraphs can be coordinated with enclosing system representations in a way that systematically exploits the predictability and efficiency of the static data flow model. This greatly enhances the power of our techniques in terms of avoiding deadlock, increasing the design space for clustering, and providing for integration with more general models of computation. We have shown the benefits of up to 95% improvement of performance by experiments. Future work will focus on optimal clustering techniques, i.e., to identify which actors should be clustered in order to minimize the scheduling overhead, hence, minimizing the number of states in the clustering FSM while maximizing the length of the static scheduling sequences. Furthermore, we have figured out in preliminary investigations that quasi static schedules allow for efficient intra processor communication synthesis as the number of tokens stored in each channel during graph execution can be predicted at runtime. As first measurements have shown promising improvements in performance, we are currently working on an automatic synthesis path for this kind of optimized software communication. Moreover, even a large cluster FSM state space has a comparatively small set of unique schedules. In future work, we will exploit this fact to generate an efficient representation of the cluster FSM state space for synthesis.

[6]The super-linear speedup in comparison to the single processor implementation is due to the different memory interfaces used in both implementations: Thanks to the multi-processor implementation, the code size for the individual processors could be reduced such that fast Local Memory Buses (LMB) can be used.

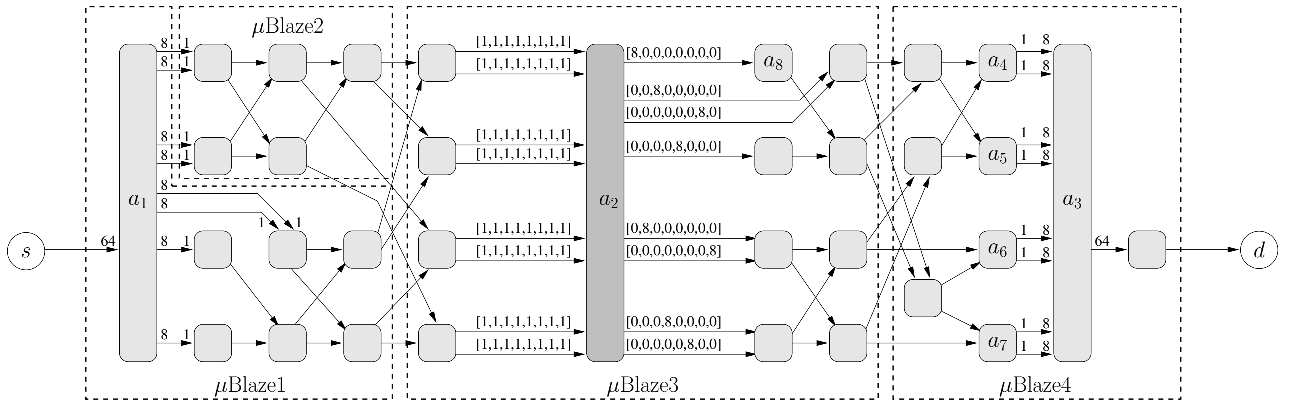

Figure 10: Clustered multi-processor implementation of the two-dimensional IDCT. Inter-processor communication is realized via hardware FIFO links.

8. REFERENCES

[1] S. Abdi, J. Peng, H. Yu, D. Shin, A. Gerstlauer, R. Doemer, and D. Gajski. *System-on-Chip Environment (SCE Version 2.2.0 beta): Tutorial*. UC Irvine, Irvine, CA, July 2003. Tech. Rep. CECS-TR-03-41.

[2] 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*, 42(3):138–150, Mar. 1995.

[3] S. S. Bhattacharyya and E. A. Lee. Scheduling synchronous dataflow graphs for efficient looping. *J. VLSI Signal Process. Syst.*, 6(3):271–288, 1993.

[4] 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.

[5] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-Static Dataflow. *IEEE Transaction on Signal Processing*, 44(2):397–408, Feb. 1996.

[6] J. T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory using the Token Flow Model*. PhD thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, September 1993.

[7] C. Choi and S. Ha. Software synthesis for dynamic data flow graph. In *Proceedings of the International Workshop on Rapid System Prototyping*, 1997.

[8] S. A. Edwards and O. Tardieu. SHIM: A Deterministic Model for Heterogeneous Embedded Systems. In *Proceedings of EMSOFT*, pages 264–272, 2005.

[9] 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*, 2007:Article ID 47580, 2007.

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

[11] A. A. Jerraya, A. Bouchhim, and F. Pétrot. Programming Models and HW-SW Interfaces Abstraction for Multi-Processor SoC. In *Proceedings of DAC*, pages 280–285, 2006.

[12] G. Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

[13] T. Kangas, P. Kukkala, H. Orsila, E. Salminen, M. Hännikäinen, T. D. Hämäläinen, J. Riihimäki, and K. Kuusilinna. UML-Based Multiprocessor SoC Design Framework. *ACM Transactions on Embedded Computing Systems*, 5(2):281–320, May 2006.

[14] V. Kianzad and S. S. Bhattacharyya. Efficient Techniques for Clustering and Scheduling onto Embedded Multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 17(7):667–680, 2006.

[15] E. A. Lee and D. G. Messerschmitt. Synchronous Data Flow. *Proceedings of the IEEE*, 75(9):1235–1245, Sept. 1987.

[16] E. A. Lee and D. G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Transactions on Computers*, C-36(1):24–35, Jan. 1987.

[17] J. L. Pino, S. S. Bhattacharyya, and E. A. Lee. A Hierarchical Multiprocessor Scheduling System for DSP Applications. In *Proc. Conf. on Signals, Systems, and Computers*, volume 1, pages 122–126, Nov. 1995.

[18] M. Sgroi, L. Lavagno, Y. Watanabe, and A. Sangiovanni-Vincentelli. Quasi-Static Scheduling of Embedded Software Using Equal Conflict Nets. In *Application and Theory of Petri Nets 1999. 20th International Conference, ICATPN'99*, June 1999.

[19] 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*, pages 9–14, 2007.