Programming and Analysing Scenario-Aware Dataflow on a Multi-Processor Platform

Reinier van Kampenhout*, Sander Stuijk* and Kees Goossens*†

*Eindhoven University of Technology, The Netherlands

{j.r.v.kampenhout,s.stuijk,k.g.w.goossens}@tue.nl
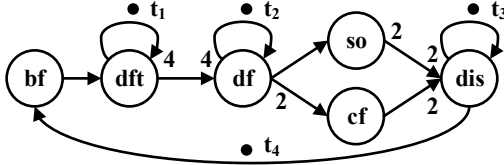
†Topic Embedded Products, The Netherlands

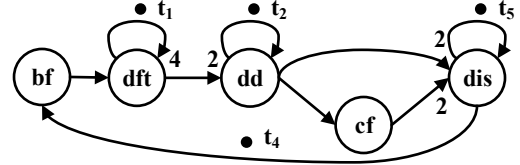Fig. 1. Scenario graph for decoding a full video frame, S_{full}.

Fig. 2. Scenario graph for decoding a delta video frame, S_{delta}.

Abstract—The FSM-SADF model of computation is especially suitable for analysing real-time applications with input-dependent behaviour such as different modes, variable execution times and scalable parallelism. Although FSM-SADF specifies which scenario transitions are possible, it does not specify how and when they are decided at runtime. Multiple actors of a scenario, e.g. video stream header parsing, may have to fire before it is known which scenario the application is in. We solve this causality dilemma with a concept for executing a sequence of scenarios, and demonstrate an implementation on multiple processors with rolling static-order scheduling. We furthermore present a platform-aware analysis model that covers concept and implementation, and integrate the contributions in a toolflow. A proof-of-concept confirms the low overhead of the implementation and the exact timing analysis of our model.

I. INTRODUCTION

Many contemporary real-time (RT) applications are streaming, meaning that data is processed when it is received rather than read on demand from local storage. The control flow of such applications often depends on the received data and can vary per iteration. Upon receiving a video frame for example, a decoder detects if it is a full frame or delta frame and calls the appropriate decoding function. In a sequential language, a programmer may solve such a dependency with a simple *if-else* construct as shown in Listing 1.

The dataflow model of computation (MoC) is a natural way to describe data-dependent behaviour [1]. In particular, finite-state machine scenario-aware dataflow (FSM-SADF) can capture different input-dependent control flows in *scenarios* [2]. Each control flow is captured in a *scenario graph* by the programmer, possibly based on existing sequential code. The scenario graph for decoding a full video frame (S_{full}) is depicted in Figure 1. When a delta frame is detected, the application behaviour and thus the scenario graph is different, see S_{delta} in Figure 2. Actor names are abbreviations of the functions in Listing 1. All allowed scenario sequences are
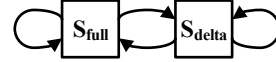

Fig. 3. The FSM of the video decoder with scenarios S_{full} and S_{delta}.

specified by the FSM in Figure 3. The graphs will be further explained in Subsection III-B.

Listing 1 Pseudo-code of an abstract video decoder.

```
1: frame = buffer_frame()
2: if detect_frame_type(frame) = full then
3:     x = decode_full(frame)
4:     sub = subtitle_overlay(x)
5: else
6:     x = decode_delta(frame)
7: end if
8: output = construct_frame(x)
9: display(output, sub)
```

While FSM-SADF allows tight analysis of applications whose control flow is input-data dependent (see Subsection III-A), it does not include an execution model. Consider the video decoder, where **bf** and **dft** are always executed first. Only after executing `detect_frame_type()` the next scenario is known. At runtime it is impossible to decide which of the scenario graphs to start with because the frame type is not yet known. Thus a causality dilemma is encountered during execution because the next scenario is detected only after the next scenario graph is partially executed. Another difference between analysis and execution concerns the consistency of state (*persistent tokens* t_i) between scenario graphs. Consider token t_2, which stores motion vectors required by both decoding functions. During temporal analysis the actual value of t_2 is irrelevant but at runtime it represents state that must
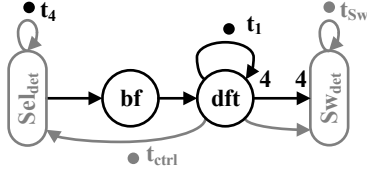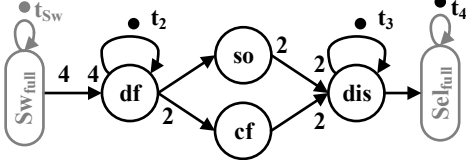
Fig. 4. Analysis graph of the detector scenario S_{det}.

Fig. 5. Analysis graph of the full frame scenario S_{full}.

Fig. 6. Analysis graph of the delta frame scenario S_{delta}.

Fig. 7. The extended FSM with detector scenario S_{det}.

be consistent between scenarios, especially after a scenario switch.

In this work we propose a concept for executing scenarios after one another (*sequencing*). Because scenarios graphs capture different behaviours of the same application, we argue that a given number of actors at the start of each scenario is identical. After executing this longest prefix graph ({$\mathbf{bf}, \mathbf{dft}$} and {$\mathbf{t_1}, \mathbf{t_4}$} in our example) the next scenario should be known. The prefix graph must be tagged by the programmer after which it is automatically split off in a *detector scenario* S_{det}, see Figure 4. After execution of S_{det} the next scenario is known and can be executed, e.g. S_{full} depicted in Figure 5. This solves the causality dilemma and is explained in detail in Subsection III-C. We extend the existing analysis model to capture the exact timing impact of this solution on the CompSOC platform. This results in a tight throughput guarantee on the execution of a sequence of scenarios.

We present the following contributions:

- a *scenario sequencing concept* for FSM-SADF that is both executable and analysable in Section III;
- a corresponding implementation for *scenario execution* in the libDataflow library in Section IV;
- a *platform-aware analysis model* in Section V.

Related work is discussed in Section II, the contributions are demonstrated in Section VI. Conclusions are presented in Section VII.

II. RELATED WORK

Several models of computation are suitable for programming real-time applications, but many of those enforce conservative assumptions when encoding input-dependent behaviour. The Synchronous Dataflow (SDF) MoC for example captures all application behaviours in one multi-rate graph [1]. There are several more flavours of dataflow [3]. For analysis the SDF[3] tool is available, which was extended in the context of this work [4]. The CAL actor language provides an implementation [5].
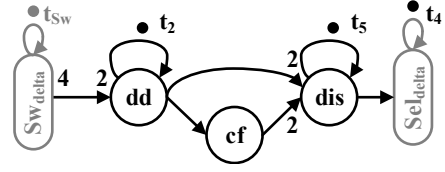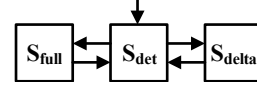
To capture different behaviours or "modes" a (partially) different graph can be selected every *iteration* (full execution) of the application. Two dataflow models support this: FSM-SADF and mode-controlled dataflow (MCDF). In the latter, the programmer captures all modes in one graph using a set of rules and special actors, namely a mode controller and *switch* and *select* actors [6]. The same graph is used both for temporal analysis and execution on the platform. Mode transitions are implicitly encoded in the mode controller. Partial graph execution is achieved by selecting a different schedule for each mode, dubbed quasi-static order scheduling (QSOS). FSM-SADF has similar expressiveness, but here the programmer creates scenario graphs with SDF semantics for each behaviour [2]. The FSM describes allowed scenario transitions. A key difference with MCDF is that no execution model is specified.

Our contribution is to execute scenario graphs directly, with minimal effort for the programmer. The analysis and execution graphs described in Sections III and V are generated automatically. During analysis our *switch* and *select* actors behave as normal SDF actors, allowing full re-use of existing analysis techniques. During execution these actors behave as their MCDF counterparts. The execution graph is reminiscent of MCDF, but the mode controller is replaced by the detector scenario [6]. Unlike QSOS, our SO schedule is extended immediately after detection of the next scenario (mode).

An existing FSM-SADF programming model splits off one-actor detector scenarios in a similar way [7]. There all actors fire but execute the encapsulated functions conditionally based on a scenario identifier token. This causes a considerable overhead in large graphs, which we avoid here. Also, it does not present an analysis model such as presented in Section V. Another programming model named disciplined dataflow networks allows to extract FSM-SADF graphs for analysis [8]. While it also leverages existing tools, it cannot execute FSM-SADF directly.

We mapped an application onto the CompSOC platform to demonstrate our concept and implementation [9]. The available libDataflow and libFIFO libraries implement dataflow execution and first-in first-out (FIFO) channels. A homogeneous synchronous dataflow (HSDF) timing analysis model is available for the platform [10].

III. SCENARIO SEQUENCING CONCEPT

A. Motivation

Static MoCs such as SDF cannot respond to input-dependent behaviour, resulting in a conservative throughput bound. Consider the code in Listing 1, where `decode_full()` or `df` has a longer WCET than `decode_delta()`. Yet due to data dependencies invisible at this level of control flow, the WCET of `construct_frame()` or `cf` might be longer for a delta frame. SDF analysis will consider the WCET of `df` *and* longest WCET of `cf` *at the same time*, although this situation can never occur. This results in an overly negative throughput bound [11]. FSM-SADF analysis considers a unique WCET for each actor in each scenario. Thus combinations that can never occur are excluded, resulting in a tight throughput bound on an applications worst-case throughput by considering the worst possible scenario sequence in the FSM [3].

Multiple application *behaviours* can be encoded in scenarios. For example, different control flows can be captured by varying the graph *topology*, thereby also changing the *mapping*. Thus the degree of parallelism in an application can be varied to deal e.g. with varying resource availability or quality-of-service requirements. Different *rates* can be applied to maintain throughput when dynamic voltage and frequency scaling is used. Modelling different *WCETs* is useful for functions such as variable length decoding for different frame types. Executing multiple scenarios after each other, including S_{det}, is the topic of this section.

B. Semantics

The full frame scenario depicted in Figure 1 is a graph with predefined topology and rates. The nodes represent the actors named $\{\mathbf{bf}, \mathbf{dft}, ...\}$, each with a known WCET. The edges represent channels, the rates at their start and end indicate how many tokens are produced and consumed each time the actors *fire* (execute). Persistent tokens indicated with $\mathbf{t_i}$ appear on some channels and are present at the start and end of each iteration of the graph.

The delta frame scenario is depicted in Figure 2, and the scenario sequences that are allowed in Figure 3. Some actors occur in both scenarios, possibly with different rates: $\{\mathbf{bf}, \mathbf{dft}, \mathbf{cf}, \mathbf{dis}\}$. Different rates may result in a different *repetition vector*, which is the number of times an actor fires each iteration. Other actors occur only in one scenario but might consume a persistent token that occurs in multiple scenarios: $\{\mathbf{df}, \mathbf{dd}, \mathbf{so}\}$.

In short, channels and actors can be connected at will in different scenarios, allowing programmers to express coarse-grained changes in the control flow. We pose a constraint in the context of this work, which is that there must be an identical subgraph that is the longest prefix graph of each scenario. This subgraph must consist of at least one actor, and the repetition vectors of the actor(s) must be identical in each scenario. After execution of the subgraph the next scenario must be known.

C. Sequence Analysis

Section I introduced a solution to the causality dilemma by splitting off the detector scenario S_{det} from the original scenarios, see Figure 4. Identification of the detector subgraph is a task of the programmer for now, all the steps described next are automated. First the FSM is transformed, S_{det} is now executed before each transformed original scenario. See Figure 7, the starting state is indicated with an incoming edge.

To model the transport of tokens from and to the detector scenario, we leverage *switch* and *select* actors borrowed from boolean dataflow. These are instantiated on the outgoing channels (switch) and incoming channels (select) at which the scenario graphs are split, and assigned a WCET of zero. See Figures 5 and 6. Each $\mathbf{Sw}$ and $\mathbf{Sel}$ receives a self-edge with a synchronisation token that has the same label in every scenario. This ensures that each synchronisation token is modeled as one physical token during analysis, as proposed in [12]. Token $\mathbf{t_4}$ is a special case. To synchronize execution of $\mathbf{Sel}$ we need $\mathbf{Sel_{det}}$ to fire before $\mathbf{bf}$. Therefore $\mathbf{t_4}$ is moved onto a self-edge and fact replaces the synchronisation token. Additionally we need one initial control token $\mathbf{t_{ctrl}}$. The analysis graphs thus generated are shown in Figures 4, 5 and 6. The $\mathbf{Sw}$ and $\mathbf{Sel}$ actors as well as newly inserted channels and tokens actors are indicated in grey.

Throughput analysis with SDF[3] uses the timestamps at which tokens are consumed and produced, the actual data value of tokens is not relevant. Temporal analysis of decoding a full video frame with SDF[3] visits the scenarios as indicated by the FSM, starting with S_{det}:

1) $\mathbf{Sel_{det}}$ fires and produces a token on the channel to $\mathbf{bf}$ as well as the new $\mathbf{t_4}$ at time τ_1;[1]
2) $\mathbf{bf}$ fires, followed by $\mathbf{dft}$ which detects scenario S_{full};
3) $\mathbf{dft}$ produces its *data* tokens on the black channel to $\mathbf{Sw_{det}}$ and a *control* token onto each grey channel;
4) $\mathbf{Sw_{det}}$ fires and produces the new $\mathbf{t_{Sw}}$ at time τ_2;

This concludes S_{det}. Note that the $\mathbf{Sw}$ and $\mathbf{Sel}$ actors did not consume time, i.e. they produce tokens immediately after consumption. Persistent tokens become available in the next scenario from the moment they are produced. We will see that this achieves synchronisation in S_{full}:

5) $\mathbf{Sw_{full}}$ is blocked until τ_2, then it fires and produces $\mathbf{t_{Sw}}$ and the data tokens towards $\mathbf{df}$;
6) $\mathbf{df}$ fires also at τ_2, i.e. immediately after $\mathbf{dft}$;
7) S_{full} continues to execute as normal until $\mathbf{dis}$ fires;
8) $\mathbf{Sel_{full}}$ fires as soon as the token from $\mathbf{dis}$ is available, producing $\mathbf{t_4}$ at τ_3.

This concludes analysis of S_{full}. Note that S_{det} can start again while S_{full} is underway, but $\mathbf{Sel_{det}}$ is blocked until τ_3. We see that the *switch* and *select* are synchronisation portals that model the transport of tokens between scenarios. The analysis of a delta frame is similar.

---

[1]No time has elapsed and the token distribution is now as in Figure 1, therefore the two starting states are equivalent regardless the location of $\mathbf{t_4}$.
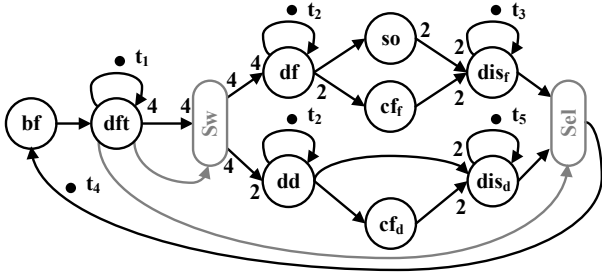
Fig. 8. The merged execution graph, the *switch* and *select* actors are indicated in grey.
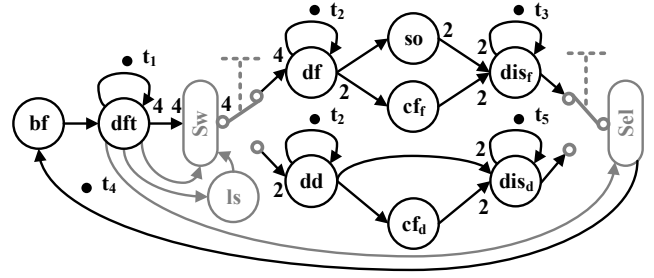

Fig. 9. The merged execution graph with implementation details, the *switch/select* and *load schedule* (**ls**) actors for a one-processor mapping are indicated in grey. The swapping of FIFO channels is indicated with the electrical symbol for a switch, also in grey. Note that this is not a valid dataflow graph.

IV. SCENARIO EXECUTION

The scenario sequencing concept presented in Section III adheres to FSM-SADF and is analysable with existing tools. The throughput analysis result of the original graphs and of the split scenarios is identical. From the implementation perspective, the split solves the causality dilemma because the next scenario graph is only started after it is detected. In this section we present an implementation for executing a sequence of scenarios that solves the following practical aspects:

A. *switch* and *select* implementation;
B. extending static-order actor schedules on-the-fly;
C. sharing persistent tokens (state) between scenarios.

A. Switch and Select Implementation

During analysis the *switch* and *select* actors ensure synchronisation but function as regular SDF actors. While scenarios are analysed separately, our implementation glues S_{det} to the other scenarios. In practice they act as multiplexer (switch) and de-multiplexer (select). For execution there is *one* detector subgraph serving both S_{full} and S_{delta} with tokens, see Figure 8. Actors **Sw** and **Sel** are indicated in grey. Synchronisation token t_{Sw} is not relevant for execution.

Such (de-)multiplexers are not available in the current libDataflow library. Consider a *switch*, whose data tokens on the input port are forwarded to one of the output ports. The rate on the other output is effectively zero. Both changing rates within a scenario and rates with value zero are currently not supported. Note however that the models from Subsection III-C fully adhere to the FSM-SADF MoC.

We propose a solution that exploits the fact that libFIFO buffers can be disconnected and reconnected without invalidating data. A *switch* actor will have a just a single output port, to which the proper channel is connected depending on the detected scenario. See Figure 9. This swapping of FIFO channels is indicated with the symbol for an electrical switch, which connects the single output port either to the channel to **df** or that to **dd**. The other channel is left unconnected on one end, effectively giving it rate zero. The FIFO swap must take place before the firing rules of **Sw** are checked, i.e. before it fires. The behaviour during runtime can be matched to the analysis steps listed in Subsection III-C as follows:

(a) the output port of **Sw** is connected to the FIFO towards **df**;
(b) **Sw** fires and consumes all of its tokens (4 in III-C);
(c) **Sw** produces tokens into the connected FIFO (5);
(d) execution of S_{full} continues as usual (6, 7, 8, 1).

Select actors are similar but demultiplex two channels to one.

B. Extending Static-Order Schedules

The libDataflow library executes dataflow graphs by iterating over the static-order (SO) schedule that is given by SDF[3] for each processor. It blocks if an actor is not ready to fire. However, the SO schedule of our proposed solution changes depending on the detected scenario. Therefore we introduce a new scheduling concept that we dub the *rolling static-order* (RSO) scheduler.

Execution starts with S_{det}, so if we were to map the decoder to a single processor the SO schedule starts with $[\mathbf{bf}, \mathbf{dft}, \mathbf{Sw}]$. After firing **dft** the next scenario is known and the SO schedule can be extended. Should S_{full} be detected, the sequence $[\mathbf{df}, \mathbf{cf_f}, \mathbf{cf_f}, \mathbf{so}, \mathbf{dis_f}, \mathbf{Sel}, \mathbf{bf}, \mathbf{dft}, \mathbf{Sw}]$ must be concatenated to the "rolling" schedule. Note that **Sel** comes at the end of S_{full}, and we immediately concatenate the next detector scenario. In this way it is ensured that the scheduler will never run out of actors to schedule. A multi-processor mapping works similarly, the only constraint we impose is that a *switch* must be mapped onto the same processor as the actor preceding it.

The RSO scheduler was implemented in libDataflow for the CompSOC platform. We initialize it with a unique SO schedule for each scenario, the start is set to S_{det}. An additional *load schedule* (**ls**) actor is inserted right after **dft** *on every processor*. See Figure 9 for the single-processor example. These **ls** actors receive a scenario token from **dft** and extend the schedule accordingly. The example schedule of S_{det} changes to $[\mathbf{bf}, \mathbf{dft}, \mathbf{ls}, \mathbf{Sw}]$. We furthermore exploit the **ls** actor to connect all FIFOs correctly before the firing rules of **Sw** or **Sel** are checked. This dependency is visualized with a grey channel in Figure 9.
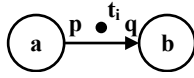
Fig. 10. Graph with two actors **a** and **b**, connected by a FIFO with production rate **p**, consumption rate **q** and persistent token $\mathbf{t_i}$.
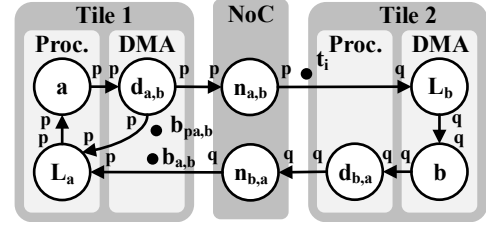

Fig. 11. CompSOC binding-aware model of the graph in Figure 10 with both actors mapped to different processors, based on [10]. **Proc** denotes processor, $\mathbf{b_{pa,b}}$ and $\mathbf{b_{a,b}}$ model the source buffer and the FIFO buffer.

C. Shared Persistent Tokens

Lastly, *persistent* tokens that appear in multiple scenarios must be consistent during execution. We solve this by instantiating the FIFO that contains such a token only once, and map the appropriate port from the source and destination actor in every scenario in which the token appears to that FIFO. Connecting multiple ports to a single FIFO is another useful property of libFIFO that avoids synchronisation issues. It leads to the requirement that all source actors of the same FIFO must be mapped onto the same processor in each scenario, and all destination actors as well. Also, multiple persistent tokens mapped onto one channel should appear in the same combination in each scenario.

V. PLATFORM-AWARE ANALYSIS MODEL

Section III introduced a scenario sequencing concept, Section IV described the implementation. To ensure correct timing analysis we re-visit the MoC and extend the existing platform-aware analysis model so that the exact timing impact of our solution is analysed. SDF³ generates different mappings from a set of storage distributions. The throughput of each mapping is analysed after expanding the mapped scenario graph to a *binding-aware graph* (BAG) [13]. This means the graph is annotated with timed models of the hardware and software platform components such as the Network-on-Chip (NoC) and the SO schedule. We will now discuss how each of the three changes from Section IV is modelled in the BAG.

The timing impact of **Sw** and **Sel** actors is modelled as follows. Firstly, the time required by the **Sw** and **Sel** actors for simply forwarding the data tokens is added to their execution time in S_{det}. Secondly, the time it takes to connect the correct FIFOs to these actors is added to the **ls** actor on that processor. The time for extending the rolling SO schedule is also added to **ls**. Thirdly, connecting multiple actors to one FIFO that holds a persistent token requires no changes but reduces the memory footprint.

SDF³ can analyse and map SDF and cyclo-static dataflow (CSDF) applications onto a CompSOC platform using an HSDF platform model [9], [10]. In the context of this work we extended the HSDF platform model for FSM-SADF analysis. The key difference is that HSDF is mono-rate, meaning that all rates on all channels have a value of one, but scenario graphs are multi-rate. Let us consider the example shown in Figure 10. If actors **a** and **b** are mapped to different processors, the channel is replaced by a combined model of the DMA, NoC and CoMik microkernel as explained in [10]. We extended that work and annotated the original HSDF model with rates as shown in Figure 11, which also shows the location of persistent token $\mathbf{t_i}$ in the model.

A consequence of multi-rate graphs is that actors can have a repetition vector larger than one. This complicates the schedule encoding in the BAG because multiple actors might be *enabled* (ready to fire) at the same time. The encoding must ensure that only the scheduled actor fires. The original HSDF schedule encoding cannot do this, so we integrated an existing technique in the FSM-SADF toolflow that can [14]. The actors modelling the DMA units ($\{\mathbf{d_{a,b}}, \mathbf{d_{b,a}}\}$ in Figure 11) follow the schedule of the actors whose FIFO they model, and must therefore be encoded using the same method. This introduces additional complexity because there can be multiple DMA actors per regular actor. We add simple dependency edges between these that follow the order in which the tokens are consumed or produced. This limits the number of enabled actors for the schedule encoding, simplifying the BAG. However, as DMA actors belonging to different regular actors on the same processor have no such implicit dependencies, the schedule encoding still considers all of those enabled at the same time. To enforce ordering, extra persistent tokens are inserted in the BAG which can slow down analysis for graphs with high rates.

Once a mapping is selected, the scenario graphs are merged together automatically which results in the graph shown in Figure 8.

VI. EXPERIMENTAL EVALUATION

A. Setup

To demonstrate the concept, model and implementation we mapped a dataflow version of the SUSAN edge detection algorithm onto a two-processor CompSOC platform [15]. Scenario S_{seq} is a sequential graph that represents the original algorithm, see Figure 12. Scenario S_{par} is a parallelized version, see Figure 13. The algorithm reads an image block by block, one of the graphs must be executed for each block. Scenarios are switched based on the image resolution, the FSM is similar to the one shown in Figure 7.

If mapped to two or more processors, scenario S_{par} can achieve a higher throughput because the computationally intensive actors **usan** and **dir** (direction) can be executed in parallel ($[\mathbf{us_1}, \mathbf{dir_1}]$ and $[\mathbf{us_2}, \mathbf{dir_2}]$). To provide both these chains with a block, **split** should have a consumption rate of two. This means the **get** (get image) actor has to execute twice. To not violate the constraint concerning the identical repetition vectors of actors in the detector scenario, the production rate on the channel from **get** to **usan** in
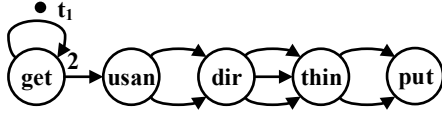
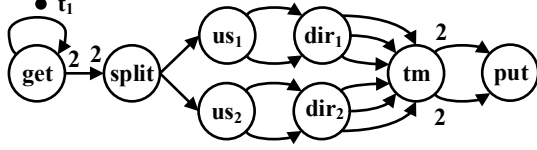Fig. 12. Scenario graph of the sequential SUSAN scenario, S_{seq}.

Fig. 13. Scenario graph of the parallelized SUSAN scenario, S_{par}.

Fig. 14. Original test image (left) and the output of SUSAN. The outer band of the image is skipped.

S_{seq} must also be two. This implies each image must contain an even number of blocks, and is an example effect of the constraints that we impose.

We instantiated a CompSOC platform with two processors, both clocked with a frequency of 100 MHz. Each processor features instruction and data memories of 256 kB each as well as one DMA with two communication memories of 16kB each. The processors are connected to each other and to an external DDR memory via a NoC [9].

B. Results

The SDF^3 tool takes the scenario graphs annotated with execution times and memory requirements as an input, plus an architecture file. The throughput is defined as the inverse of the number of cycles required to complete one iteration of the application. The analysis gives a maximum throughput of $2.7 \cdot 10^{-7}$ for SUSAN. We measured a throughput of $3.1 \cdot 10^{-7}$ when executing the application on the platform using the implementation described in Section IV. The fact that the actual throughput is a little higher than the throughput given by the analysis shows that the model is conservative. The graphic output of the SUSAN edge detection algorithm is shown in Figure 14.

A *load schedule* actor takes 365 cycles to execute including the wrapper, the *switch* actor takes 1635 cycles. This brings the timing cost of our libDataflow modifications to 2000 cycles, which are accounted for in the analysis model. We argue that the impact of scenario sequencing on the applications timing is minimal, as the total WCET of SUSAN is 3.3 million cycles. The size of the library is increased by 3 kB.

VII. CONCLUSION

A causality dilemma is encountered when attempting to execute a sequence of FSM-SADF scenario graphs. In this work we propose a concept and implementation for executing a sequence of scenarios that solves this dilemma with minimal effort for the programmer. A detector prefix graph must be tagged, after which the scenarios are automatically split and annotated for analysis. The resulting platform-aware model captures the exact timing impact of the implementation, and fully adheres to FSM-SADF.

We implemented the concept in the libDataflow library. The rolling static-order scheduling method extends the SO schedule each time a scenario is detected. Mapping the SUSAN edge detection algorithm onto a two-processor platform resulted in a throughput that is slightly higher than the bound given by the analysis. This shows the analysis model is both conservative and precise, while the cost in terms of timing and memory footprint is marginal. The overall approach allows to capture input-dependent applications in FSM-SADF and execute them on multi-processor hardware in an automated manner.

REFERENCES

[1] E. Lee and D. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Transactions*, 1987.

[2] B. D. Theelen et al., "A scenario-aware data flow model for combined long-run average and worst-case performance analysis," in *MEMOCODE*, 2006.

[3] S. Stuijk et al., "Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications," in *SAMOS*, 2011.

[4] ——, "SDF^3: SDF For Free," in *ACSD*, 2006.

[5] J. Eker and J. W. Janneck, "CAL Language Report: Specification of the CAL actor language," UC Berkeley, Tech. Rep., 2003.

[6] O. Moreira and H. Corporaal, *Scheduling Real-Time Streaming Applications onto an Embedded Multiprocessor*. Springer, 2014.

[7] R. van Kampenhout et al., "A Scenario-Aware Dataflow Programming Model," in *DSD*, 2015.

[8] F. Siyoum et al., "Automated extraction of scenario sequences from disciplined dataflow networks," in *MEMOCODE*, 2013.

[9] K. Goossens et al., "Virtual Execution Platforms for Mixed-time-criticality Systems: The CompSOC Architecture and Design Flow," *SIGBED Rev.*, 2013.

[10] A. Nelson et al., "Dataflow Formalisation of Real-time Streaming Applications on a Composable and Predictable Multi-Processor SOC," *JSA*, 2015.

[11] S. V. Gheorghita et al., "Automatic Scenario Detection for Improved WCET Estimation," in *DAC*, 2005.

[12] F. Siyoum et al., "Worst-case Throughput Analysis of Real-time Dynamic Streaming Applications," in *CODES+ISSS*, 2012.

[13] S. Stuijk et al., "Multiprocessor Resource Allocation for Throughput-constrained Synchronous Dataflow Graphs," in *DAC*, 2007.

[14] M. Damavandpeyma et al., "Modeling static-order schedules in synchronous dataflow graphs," in *DATE*, 2012.

[15] S. Smith and J. Brady, "SUSAN—A New Approach to Low Level Image Processing," *IJCV*, 1997.