A Scenario-Aware Dataflow Programming Model

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

***Abstract*—The FSM-SADF model of computation allows to find a tight bound on the throughput of firm real-time applications by capturing dynamic variations in scenarios. We explore an FSM-SADF programming model, and propose three different alternatives for scenario switching. The best candidate for our CompSOC platform was implemented, and experiments confirm that the tight throughput bound results in a reduced resource budget. This comes at the cost of a predictable overhead at run-time as well as increased communication and memory budgets. We show that design choices offer interesting trade-offs between run-time cost and resource budgets.**

I. Introduction

Multi-core platforms have become ubiquitous in embedded systems. Such platforms offer increased throughput as multiple tasks can be executed simultaneously. To exploit this and to reduce the number of devices in a system, multiple applications can be merged onto one multi-core processor. When applications have real-time requirements, the execution time of each application must be isolated from interference by others. It has been shown that multiple applications of which some have firm real-time (FRT) requirements can be merged onto a platform if strict cycle-level composability is supported in the hardware and system software. This is the case for platforms that use time-division multiplexing (TDM) to isolate access to shared resources [1], [2]. The number and type of resources reserved for an application determines its worst-case execution time (WCET). If analysis provides tight bounds, fewer resources are needed and more applications can be merged onto the same platform. Dynamic application behaviour is a complicating factor and leads to a WCET that is much longer than the average execution time.

If we take for example an MPEG decoder that decodes I-frames and P-frames, the analysis must consider the frame that takes the most time to decode, even if the amount of frames of that type is only a fraction of the total. This results in unused reserved resources. Moreover, if one application implements both frame decoders, the timing model captures a worst-case behaviour that can never occur in reality. The first problem can only be solved by dynamically changing resource reservations, which makes it hard to give FRT guarantees. Instead we focus on the second problem, which can be alleviated through a model of computation (MoC) with an associated programming model, toolflow, and platform that take dynamic variations into account at the source code level. If such events are captured in the MoC, analysis can account for them to derive tight bounds for each case. Finite-state machine scenario-aware dataflow (FSM-SADF) does exactly that [3]. This MoC splits an application into scenarios that each describe a distinct behaviour. The separate analysis of these scenarios results in a tighter overall bound on the throughput and thus in a better resource allocation than other dataflow models [4].

Programming models such as CAL support some form of data-driven switching at run-time, but offer no natural way to express and implement FSM-SADF programs [5]. In this work we propose a programming model that implements FSM-SADF. It allows intuitive design of scenarios, and applications can be analysed and mapped with the publicly available SDF^3 tool [6]. We explore different implementation alternatives and select the best candidate for our composable CompSOC platform. It is implemented in the form of a platform library that supports run-time scenario detection and switching. Experiments show that no cycles are required for scenario switching and that there are interesting trade-offs between run-time cost and resource budgets. Overall, we argue that the programming model is suitable for dynamic applications because it avoids over-reservation of resources by responding to data-driven events.

The outline of the paper is as follows. Related work is presented in Section II. The MoC and platform are detailed in Section III. A problem analysis is given in Section IV, different solutions are explored in Section V. The implementation is presented in Section VI, experimental results in Section VII. We conclude with Section VIII.

II. Related Work

A. Models of Computation

To support FRT (streaming) applications on multi-core processors, a MoC should support intuitive parallelization while providing tight WCET bounds through automated analysis. One way of achieving this is time-triggered (TT) design-time static scheduling of an application's tasks [7]. A drawback of static scheduling it that it is non-work conserving, making it difficult to respond to events at run time. In fact, the system always experiences the worst-case execution, which inherently leads to a waste of resources. Giotto is a TT programming model which introduces platform independent timing constructs to create a schedule for TT architectures [8].

A different MoC that supports FRT is dataflow [9]. Dataflow graphs naturally display parallelism, as the actors and channels essentially describe the control flow of a program. Example implementations of dataflow are the CAL actor language and the SDF^3 tool [6], [10]. There are many dataflow flavours, each of which differs in expressiveness, implementation efficiency and analysability [4]. A number of variants are analysable for FRT systems, and are work-conserving when combined with the proper scheduling algorithm. However, it is still hard to account for dynamic behaviour within an application. One method proposed to express such variability is mode-controlled dataflow (MCDF) [2]. When the designated mode controller fires (executes), only a pre-defined subgraph of the MCDF graph is executed based on the mode control

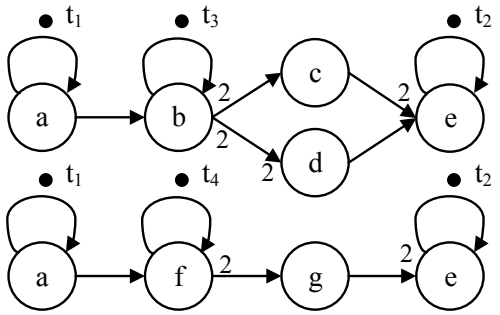

Fig. 1. Scenarios S_1 (top) and S_2 of the example application.

output of the controller that is connected to all actors. FSM-SADF is a similar model [3]–[5], [11]. Because it is flexible but can still be properly analysed, we consider FSM-SADF the best candidate to achieve an efficient work-conserving parallelization.

B. Composable Platforms

In multi-core processors, an access conflict arises when tasks on multiple cores access a shared resource simultaneously. Many solutions to this problem have been proposed, most of which rely on resource reservation. We define composability as isolated *actual-case* timing behaviour, meaning that a running application is not influenced by another application even by a single cycle. The Time-Triggered Architecture (TTA) is a composable platform on which every thread and every packet is statically scheduled according to a global clock, fitting to the TT MoC [7]. A working MP-SoC prototype featuring a TT network-on-chip (TTNoC) has been presented [12]. A drawback is that it is difficult to distribute a global clock to all parts of the system in contemporary chips. The TTNoC solution is to provide a coarse global time base on which to synchronize. Building upon the TT concept is the Precision Timed Machine (PRET). It proposes a micro-architecture, compiler and programming language with extensions that should give complete control over the timing behaviour, but has not been fully implemented yet [13]. The CompSOC platform has a TT-NoC, inter-task multi-processor scheduling and supports both the TT and dataflow MoCs for firm-, soft- and non-real-time applications [1]. We used CompSOC for this work and describe it further in III-B. A number of research projects focus on developing platforms that have composable properties, notably parMERASA [14], T-CREST [15] and MultiPARTES [16].

III. Background

A. FSM Scenario-Aware Dataflow

An FSM-SADF application consists of multiple *scenarios* of the application. Each scenario describes a specific behaviour, and is expressed in a graph consisting of actors, channels, initial tokens, and rates. Two example scenario graphs are shown in Figure 1. Each graph consists of actors (nodes) that communicate tokens through channels (directed edges), that each have a production and consumption rate. Rates are shown at the in- and output ports, but are omitted when 1. Tokens are units of data and are depicted as black dots if they are initially present. The rate determines how many

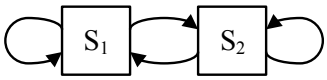

Fig. 2. Finite-state machine of the example application.

tokens are produced on or consumed from a channel when a producer or consumer actor fires. An actor fires if sufficient input tokens are available on its incoming channels. Upon firing it consumes all input tokens, executes, and produces all output tokens. A channel whose source and destination are the same is called a self-edge. Actors are stateless, that is, no data is conserved between subsequent firings. Self-edges are used to model state and to limit multiple simultaneous actor firings (auto-concurrency). They have an initial token labeled t_i. All initial tokens are *persistent*, which means that they must be stored until the next firing of that actor. In an *iteration* of a graph, actors are fired (at least once) until all tokens have returned to their starting position. If the actors are firing we say the graph is *executing*, if all tokens returned we say the iteration has *completed*. In the example, actor c and g will fire two times per iteration, and we say that their *repetition vector* entry is two.

The example consists of two scenarios, each of which may be executed indefinitely. The possible transitions between scenarios are specified by the FSM shown in Figure 2. Some actors, channels, initial tokens, and rates are different and some are the same. Actors a and e appear in both scenarios and have a self-edge with the same initial token (i.e., labeled t_1 resp. t_2). Scenario selection is data dependent; in this example actor a is the designated scenario *detector*. When a has fired, it indicates which scenario is active next. If a scenario executes, there is nothing that stops a from firing again as long as there is enough space on its outgoing channels. When a detects a scenario other than the current, a *scenario switch* occurs. That scenario graph starts executing, which means one or more scenarios can be executed in a *pipelined* manner. Synchronization of persistent tokens (i.e., initial tokens with the same label that appear in different scenario graphs) is a concern we will explore in Section IV.

B. CompSOC Platform

In this work we use the existing CompSOC platform, which features cycle-accurate composability by means of virtualization [1], [17]. Each application is assigned its own *virtual platform* (VP). Each VP consists of a set of resource budgets for core computation time, NoC capacity, and memory capacity and space. A VP can have time budgets on multiple cores and memories, and each resource is shared and can host multiple VPs at the same time. Within a VP an application has full control over its resources. VPs themselves are scheduled using time division multiplexing (TDM) scheduling, each VP is assigned a number of time slots. The TDM slot limits are enforced with the precision of a single clock cycle by the CoMik microkernel [18]. This guarantees cycle-accurate composability between VPs, and enables independent design, verification and execution of applications.

A *bundle* is an application encapsulated in a VP. Bundles can be loaded composably at any time in a running system [19]. Loading a bundle consists of a number of steps. First the VP is created, i.e. resources are reserved according to the

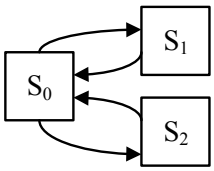

Fig. 3. Extended FSM with detection scenario S_0.

budget descriptors. Then the code (actors) and data (initial tokens) are loaded into the freshly reserved memories of all cores on which the application is mapped. After that the FIFO channels and POSE operating system are initialized [20]. Channels are in local memory when both producer and consumer are on the same core. If they are on different cores however, a channel is set up across the NoC and the FIFO is placed in a communication memory of the consuming core or any other shared memory. POSE supports both round robin and static order (SO) scheduling of dataflow graphs, in this work we use the latter. The SO scheduler iterates over a list of actors and checks the firing rules of each of them. If the rules are satisfied, the actor fires. If not, the scheduler blocks until they are.

IV. Problem Analysis

A. Scenario Detection

Two main challenges are addressed in this work: the programming of scenarios, and the detection and execution of different scenarios on a composable platform. FSM-SADF specifies each scenario as a dataflow graph that may share actors and tokens with other scenarios. Together with the FSM, this is sufficient to compute the WCET. Both the dataflow graphs and the FSM must be implemented to obtain an application executable. Different types of graphs have been implemented in previous work [2], [5], [21]. The challenge is in programming an FSM that specifies the order of the scenarios. A straightforward approach is to keep the scenario graphs and FSM separated, and implement the latter as a decoupled control entity. As scenario detection is data-dependent however, this controller must perform computations to arrive at its decision. If we take for example an MPEG decoder, the detection of an I or P frame depends on the result of the variable length decoding (VLD). After detection the controller can start the correct scenario, which *also* needs information from the VLD. If control were to be strictly separated from the data path, the VLD must be executed again, leading to unnecessary redundant computations. In other words, the VLD actor is contained in the controller and both scenarios. For each frame both the controller and one of the scenarios are executed.

We argue that the separation between the FSM and scenarios is a powerful formal abstraction, but not suitable for an efficient implementation. Instead we propose to *intertwine* the implementation of the FSM and scenario graphs. While scenario detection is hidden in the original FSM, we make it explicit by creating an additional "detection scenario" for the scenario detector actor. The extended FSM for the example from Figure 1 is depicted in Figure 3. A scenario S_0 that contains actor a is added, and that actor is removed from the two other scenario graphs. We see that S_0 has two possible successors that depend on the data: either scenario S_1 or S_2 is executed. The implementation of all scenarios is deterministic,

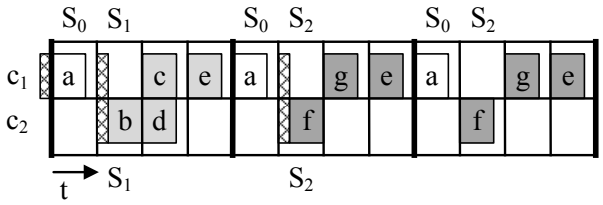

Fig. 4. Two-level schedule of the example. The TDM schedule consists of four slots, separated by horizontal bars, and is repeated three times. Inside each slot, the scenario SO schedule is executed.

but the switch after S_0 is possibly non-deterministic. We will for now only consider graphs with one detection scenario. In the MPEG decoder example, the VLD would be encapsulated in a detection scenario and the I- and P-frame decoders in two separate scenarios. Thus the VLD is only executed once, followed by one of the decoders.

B. Timing and Execution

To see what difficulties regarding timing arise during execution, let us go through an example implementation on a dual-core system step-by-step. To illustrate dynamic loading, we choose to wrap each scenario in a separate bundle as if it were a stand-alone application. Detection scenario S_0 executes every iteration, and is never removed after being loaded. As soon as it detects either S_1 or S_2, that bundle is loaded and starts executing. If it detects a scenario that is currently active, that scenario can remain in memory and execute in a pipelined fashion. Otherwise, the current scenario is removed and the other scenario is loaded. Thus, pipelining of different scenarios is not possible, which negates some of the benefit of FSM-SADF.

Actors $[a], [c, e], [g, e]$ are mapped onto core c_1, and $[b, d], [f]$ on c_2. These ordered sets also give the static order schedules. The VPs that encapsulate the scenarios are scheduled using a four slot TDM schedule on each core. The VM of S_0 is scheduled in the first slot on c_1, the other three slots and all four slots on c_1 are for the detected scenario. A timing diagram for the example application is depicted in Figure 4. At startup, S_0 is loaded into memory, indicated by the slice with pattern on the very left. Then the TDM sequence starts, indicated by the fat vertical black bar. The first slot on c_1 is assigned to S_0, actor a fires and detects scenario S_1, which is scheduled in the next slot. First the scenario bundle is loaded on both cores (slices with pattern), after which actor b fires on c_2. In the next slot, c fires in parallel to d. Then e fires and the iteration is complete. The TDM schedule starts over again, now S_2 is detected by a. Both cores are reconfigured, note that persistent token t_3 must be stored for later use and token t_2 must be transferred to S_2. Then $\{f, g, e\}$ can fire. The next firing of a detects S_2 again, which starts another iteration of the same scenario without reloading. Pipelined execution of this scenario would have been possible had a fired earlier.

While the duration of actor firings and bundle loading times are idealized in the figure, they will greatly vary in reality. An actor firing may take much longer or shorter than the duration of a slot. Because the SO schedules of all scenarios are decoupled, these timing differences can lead to inefficiencies. For instance, detector a could fire multiple times in its slot, only bounded by the capacity of its outgoing channels to b and

TABLE I. OVERVIEW OF TRADE-OFFS BETWEEN ALTERNATIVES.

Alt.	One VP per	Graph iterations	Switch duration	VP size
A	Scenario	Until switch	Bundle load or VP crossing	$S_0 + (S_1 \cup S_2)$
B	Application	Unlimited	Reconfiguring rates/schedule	$\sum S_i$
C	Iteration	1	Bundle load	$S_0 + nS_1 + mS2$

f. If these channels are full, a can not fire and the slot reserved for S_0 is unused. If the channel capacity is high, it can fire often and there can be a large delay between the detection and execution of a scenario. Conversely, if a does not finish before the end of its slot, no work can be done in the following three slots *at all*. We conclude that execution can only be efficient if the composition of TDM slots and SO schedules is balanced at design time. That can only be achieved if the execution time of actors is tightly bounded, which is a strength of FSM-SADF.

Another observation is that tokens must progress from S_0 to S_1 and S_2, and persistent token t_2 must commute between S_1 and S_2. This requires a modification of POSE and the C-HEAP FIFO library [18]. The time required to transfer tokens must be accounted for in the analysis of one of the scenarios or in the OS. If we would allow S_1 and S_2 to be kept in memory at the same time, pipelining of multiple scenarios becomes possible while the resource budget would increase. This however creates an additional dependency on persistent token t_2, as one scenario might overtake another.

The switching time between scenarios is equal to the bundle loading time if a scenario other than the current is detected. When the same scenario is detected, the loaded scenario can be executed again and the switching time is reduced to transferring t_2 and the token produced by a to the other VP. It can be derived from [19] that it takes around 21 cycles on average to load one byte of object code. That time is both bounded and predictable. Thus this implementation offers a trade-off between the switching time and the increase in throughput offered by FSM-SADF.

In the example, the schedule results in simultaneous loading of the bundles on both cores. This might not be the case for other applications. If bundles are loaded on different cores at separate moments, channels that go across the NoC can be temporarily disconnected. This is not foreseen in the current FIFO implementation. If d would finish sooner than c for example, c_2 could be reconfigured while e did not consume the token from d yet. The producer has disappeared while the consumer has still not fired. All these fundamental issues must be addressed in an implementation of the FSM-SADF programming model.

V. EXPLORING IMPLEMENTATION ALTERNATIVES

From the analysis we conclude that the crux of the implementation is scenario switching. We will now explore three alternative implementations of the FSM-SADF programming model. They vary in the way scenarios are distributed over virtual platforms. One extreme is to pack all scenarios in one VP, the other to load a new VP for each iteration. Table I summarizes the alternatives.

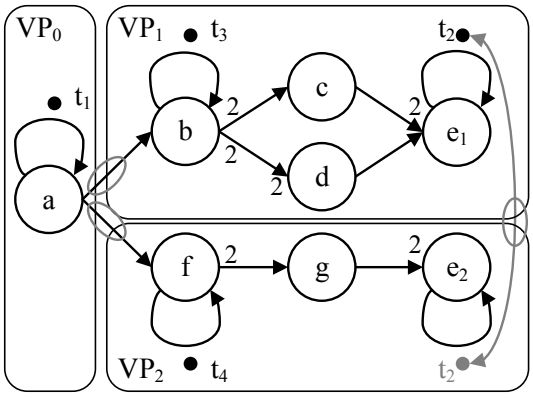

Fig. 5. One VP is loaded upon a scenario switch, and is swapped when another scenario is detected. VP crossings are encircled.

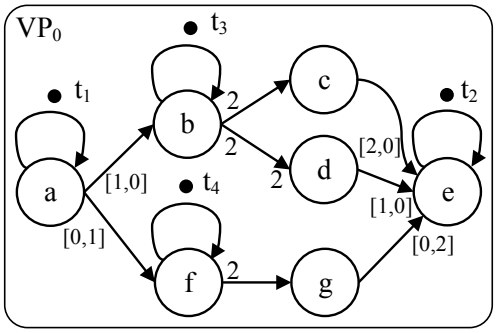

Fig. 6. All scenarios are contained in one VP, a detection scenario is not necessary.

A. VP per Scenario

The first alternative has already been discussed in section IV. Each scenario is packed into a VP, and loaded as soon as it is detected in S_0. Figure 5 shows the distribution of scenarios over VPs. When another scenario is detected, the current VP is "paused" as soon as the iteration completes and its persistent tokens stored. The VP can then be removed and swapped for another. The resource budgets that must be reserved for the application consist of the union of the budgets for each scenario. We assume most resources can be re-used in all scenarios. Still, this is another detail that must be taken care of in the implementation

The switching time is equal to either the bundle loading time or the transfer of tokens to the next scenario. As analysis must consider the worst-case, the throughput will be based on the former. The trade-off between the throughput gain of FSM-SADF and cost for switching must be carefully considered. If we assume a realistic bundle is at least 10 kB, loading will take hundreds of thousands of cycles [19]. Besides the scheduling complications and FIFO implementation issues, this is yet another drawback. Let us therefore look into a variation already touched upon, where all scenario VPs are kept in memory at the same time.

B. VP per Application

This alternative assumes all scenarios are loaded at all times. We choose to merge both graphs and encapsulate them in one VP to save resources. In this case not only scenario

detector a but also actor e can be merged, see Figure 6. This requires variable rates on the outgoing edges of a and incoming edges of e, note that this is not a standard dataflow graph. In each iteration a writes to one of the channels with variable rate, triggering execution of the corresponding scenario graph. The rates are denoted as $\{S_1, S_2\}$ on the edges. In e the scenarios merge again, the rates must be set correctly to fire when tokens from the current scenario arrive. The platform supports variable rates.

Scheduling can now be taken care of within a VP, which avoids hierarchical scheduling problems. While it is possible to derive a deadlock-free SO schedule for each scenario, both the SO schedule and rates must be changed after scenario detection. If S_1 is detected for example, schedules $[c, e, a]$ and $[b, d]$ must be loaded on c_1 and c_2 respectively. The POSE run-time SO scheduler must be extended to allow such dynamic loading of schedules. Moreover, a cannot fire again before e has fired, which rules out pipelining. The switching overhead now consists of the time required to set up the channel rates and SO schedule. A solution that circumvents both variable rates and dynamic loading of schedules works as follows. *All* actors could be fired each iteration, each of which only executes the actor function if belongs to the detected scenario. We will explain this in more detail in section VI. Thus modified, this alternative *does* allow pipelining scenarios.

Persistent tokens can be dealt with in the following way. We impose the restriction (for all alternatives) that persistent tokens may only occur in multiple scenarios if they are on channels with the same producer and consumer. In that case, these actors and the persistent token can be merged, see e and t_2 in the example. Tokens that appear in one only scenario pose no problem. The resource budgets of this alternative are larger compared to that of alternative A. Firstly, all actors must be loaded in local instruction memory. The exact size difference depends on the application, but remember that the union of the scenario resource budgets must be considered in alternative A. Secondly, all channels are always present in the data and communication memories. Note however that all tokens on self-edges must be stored in a (remote) memory in the solution from Section V-A as well. This alternative trades resource-budgets against switching time. Minor modifications to the OS and FIFO library are necessary.

C. VP per Iteration

The third alternative is another variation on alternative A, now a new VP is loaded for each scenario *iteration* and executed only once. The major advantage over solution V-A is that different scenarios can be pipelined. However, the switching time is now always equal to the time required for loading a bundle. It could be reduced by having one instance of each scenario on standby, at the cost of an increased resource budget. The overall budget must consider the footprint of the maximum number of scenarios that can be pipelined. In the example, this can be nS_1 and $mS2$, where n and m must be found through graph analysis.

D. Trade-offs

An overview of the alternatives discussed in this section is given in Table I. The choice between the first two depends on both the application and the platform. For applications with large, dissimilar scenarios that are not switched often, alternative A may be the best choice. It reduces the resource budgets, especially those of instruction memory, at the penalty of a longer switching time. The implementation effort of this alternative is considerable, as tokens must be transferred between scenarios and scheduling is inefficient if the SO and TDM schedules are not properly balanced. Applications with that fit into the instruction memory and have frequent scenario switches will benefit from alternative B. The resource budgets must accommodate the complete application, but switching time is minimal. We suggested a more optimal implementation that avoids variable rates and run-time loading of schedules. The third alternative allows the same level of pipelining as the second, but has a longer switching time *or* a higher resource budget and will not be considered further. Inside each alternative, minor design choices offer a trade-off between switching time, resource budget and implementation cost. For real-time streaming applications we argue that alternative B is the best choice, and implemented it for this work. The decisive factors is the low switching time which is key for profiting from FSM-SADF. The high cost for switching of the first alternative can only be amortized by huge reductions in the throughput bound, which seems unlikely.

VI. Implementation

We implemented the FSM-SADF programming model based on alternative B described in Section V. Our solution consists of an intuitive method to write down FSM-SADF applications, an extension of the existing toolflow with an algorithm to merge scenarios into one executable graph, and a run-time extension of the CompSOC platform to support execution of the graph [22].

A. Programming Model

For the development of FSM-SADF applications we propose a two-way programming scheme. The first step requires the identification of scenarios in a dataflow graph. Tools such as benchmarking suites may help, but the essential design decisions are made by the developer. Creating too many scenarios leads to a large overhead in terms of resource budgets or switching time, too few might negate the benefits of FSM-SADF. Once scenarios are identified, the resulting scenario graphs and FSM must be written down in the existing XML notation scheme of SDF^3. The resulting graphs for the example application are given in Listing 1. Note that the listing is greatly simplified, both in length (the original file is 285 lines) and detail (e.g. port numbers on channels are omitted). The programmer has manual control over every detail, but can apply a script to generate repetitive structures such as actors and channels. Further note that it is not necessary to add detection scenario S_0, because the FSM is implemented by merging all scenarios later on. The only addition to the existing SDF^3 XML format is the scenario detector label, see actor a in Listing 1. The second step is programming the bodies of the dataflow actors in C, taking advantage of the POSE operating system and libraries provided with the platform. Each actor body can consist of multiple functions, the in- and output arguments of which must be mapped to actor ports, see Listing 2.

Listing 1 Simplified notation of scenario graphs and FSM.

```
<applicationGraph name="smap">
 <scenariograph name="scenario1">
  <actor name="a">
   <port name="fab_0"rate="1"type="out"/>
   ...
  </actor>
  ...
  <channel name="fab14"srcActor="a"dstActor="b"/>
  ...
 </scenariograph>
 <scenariograph name="scenario2">
 ...
 </scenariograph>
 <fsm initialstate="s1"detector="a">
  <state name="s1"scenario="scenario1">
   <transition destination="s1"/>
   <transition destination="s2"/>
  </state/>
 ...
 </fsm>
</applicationgraph>
```

Listing 2 Pseudo-code to connect function arguments to actor ports.

```
fifo_in = os_get_data_in_ptrs()
fifo_out = os_get_data_out_ptrs()
function_body(fifo_in, fifo_out)
```

B. Toolflow

The scenario and FSM are the input to the SDF^3 toolchain, together with a description of the architecture. SDF^3 is extensively documented and can analyse the throughput of FSM-SADF applications [6]. It supports code generation for the CompSOC platform, and generates multiple scheduling and binding solutions [1]. The developer selects one of these, after which the "scenario-merge" transformation is applied, that was newly developed for this work and first described in Section V. It *merges* the scenarios to obtain a single graph that can be encapsulated in a VP as described in Section V-B. The pseudo-code of the algorithm merging the scenario graphs is given in Listing 3. The algorithm for merging the channels is similar and can easily be derived.

The arguments of all functions are formatted as *(source, destination)*. The first scenario graph is copied into the new *merged* graph after which we iterate over all actors of all other scenarios. If the actor is already in the *merged* graph, we add any ports that it does not yet have, and otherwise we add the whole actor. The graph resulting from executing "scenario-merge" on the example application is the one depicted in Figure 6. As noted, execution is not trivial as some channel rates are data-dependent and can be zero. The rates and SO schedule should be changed after scenario detection. Consider for example a switch from S_1 to S_2. Actor a detects the switch and should write its tokens on the channel to f rather than to b. To do that, the production rate of a on that channel should temporarily be made zero. Then the SO schedule of S_2 should be loaded.

We propose a solution that deals with both these problems

Listing 3 Pseudo-code of the scenario-merge algorithm.

```
merged=scenario[0]
for i = 1 to num_scenario−1 do
  for j = 0 to scenario[i].num_actors−1 do
    current=scenario[i].actor[j]
    new=find_actor(current, merged)
    if !new then
      add_actor(current, merged)
    else
      for k = 0 to current.num_ports−1 do
        if !find_port(current.port[k], new) then
          add_port(current.port[k], new)
        end if
      end for
    end if
  end for
end for
```

at once by firing *all* actors in *each* iteration, even if they do not belong to the detected scenario. In this way, both the rates and schedule do not need to be changed. We modify the run-time wrapper that encapsulates actors to implement this scheme. When a detector actor has detected the scenario, its wrapper will forward scenario identifier to *all* succeeding actors. It also produces all tokens specified by the rate on the outgoing channels, even if empty tokens must be inserted. The wrapper of each non-detector actor inspects the scenario identifier before executing the actor body. If the actor is part of the indicated scenario, the actor body is executed. If the actor does not belong to the detected scenario, the rest of the token(s) is empty and the body is not executed. Either way, the actor produces all tokens on each outgoing edge according to the rate, even if they are empty. Tokens on self-edges are automatically preserved if not overwritten. This solution preserves standard dataflow behaviour and is platform-independent. No special edges are necessary, and scenarios can be pipelined without ever overtaking one another.

The challenge is to send the scenario identifier from the detector to the succeeding actors. Numerous solutions are possible. Firstly, an extra channel with rate one could be added from the detector to each actor. This adds as many FIFOs as there are non-detector actors, which is especially expensive if these are mapped to different cores. A second option is to re-use existing FIFOS and extend all tokens with a scenario identifier field. Although this leaves the graph and rates intact, it is wasteful if tokens are small and rates high. Thirdly, we can also re-use the channels by increasing the rate with one. This is wasteful if tokens are large, and becomes cumbersome for repetition vectors larger than one. A fourth option is to piggy-back scenario identifiers in existing FIFOs without changing the rates. Then the scenario can be transferred only once per iteration, and no space is wasted. However, this has implications on the FIFO buffer sizing and must be consistent across the toolchain. It also requires fundamental changes to the FIFO library which makes it unfit for other dataflow flavours. The last three options can be optimized by calculating a (minimum) spanning tree of the dataflow graph and sending the tokens over those channels only.

We selected the second option because it is straightforward and is compatible with other dataflow types. The changes to the

merged scenario graph are minimal, the token size on channels that are not self-edges must be increased by one word to facilitate a scenario identifier field. The obvious disadvantage of sending the scenario many times with small tokens and high rates can be negated with a simple optimization. In such cases the greatest common denominator (gcd) of the production and consumption rate can be calculated. If both rates are divided by the gcd and the token size is multiplied with the gcd, the same data can be transferred with reduced overhead.

C. Platform

At run-time our FSM-SADF implementation executes as a SDF graph, except for three new system calls in the actor wrappers. The pseudo code for a wrapper is given in Listing 4. The first new system call is *os_get_scenario*, which extracts the scenario identifier from the incoming channel(s). Obtaining pointers to the in- and outgoing FIFOs are obtained is already part of POSE. The next new system call *os_in_scenario* checks if the actor is in the current scenario. If it is, the function body is executed. Finally the new *os_set_scenario* call forwards the scenario on all outgoing non-self edges.

Listing 4 Pseudo-code of a wrapper that contains a regular actor.

```
1: scenario = os_get_scenario()
2: fifo_in = os_get_data_in_ptrs()
3: fifo_out = os_get_data_out_ptrs()
4: if os_in_scenario(scenario) then
5:     function_body(fifo_in, fifo_out)
6: end if
7: os_set_scenario(scenario)
```

The wrapper for a scenario actor is slightly different, as the scenario identifier is produced by the actor body. This means only the call to *os_set_scenario* is necessary. All wrappers are generated by the toolflow, together with the instantiation and memory map of all FIFOs and code to initialize CoMik and POSE. The implementation of *os_get_scenario* and *os_set_scenario* system calls is straightforward: they need to iterate over all consumed respectively produced tokens in each channel.

VII. Experimental Results

We evaluated our implementation regarding resource budget requirements and run-time overhead by considering two test setups on a duo-core platform. The first comprises the CoMik hypervisor, modified POSE OS, and the example application described in the previous sections. Actor bodies are filled with NOP instructions to simulate a computational load. The application switches between scenarios every iteration. Actor d is mapped to core c_2, all other actors to c_1. The second setup is similar, but with the original POSE and no scenario switching, so all actor bodies are executed each iteration. All timing measurements are cycle-accurate, the calling overhead of which was subtracted from the results. The experiments are expected to show the exact difference between our FSM-SADF programming model and the unmodified platform. Firstly, we measured the time between the exit of each actor wrapper and the arrival in the next. This does *not* include the time required for any system calls in the wrapper. We measured no timing difference between the two setups. This shows that the POSE modifications do not affect the scheduling of actors. More importantly, it shows there is no time penalty for a scenario switch. Hence, the switching time of our implementation is zero.

TABLE II. Overview of resource budgets.

Setup	TDM slots	I-mem [bytes]	D-mem [bytes]
FSM-SADF	$1+6$	5072	1525
SDF	$2+7$	4040	1310
S_0	$1+1$	577	187
S_1	$3+1$	2308	748
S_2	$6+1$	1731	561
$S_0+(S_1 \cup S_2)$	$7+2$	2885	935

Next we determined the static computation cost of our proposed solution. As shown in Listing 4, three new system calls are made in each regular wrapper. The *os_in_scenario* system call iterates through an array with the actor identifiers of the given scenario. Thus, the overhead depends on the number of actors in the scenario. We measured a static component of 72 cycles for the function call plus 60 cycles per array entry. The *os_get_scenario* function is more complex, it iterates over all incoming channels and checks if it a self-edge. These must be skipped, as they do not contain a scenario identifier. The scenario identifier of the consumed tokens of all other channels is inspected. We measured a static component of 148 cycles plus 38 to skip over a self-edge, 55 for each non-self FIFO, and 57 per token. For example, this system call costs $148 + 38 + 55 + 2 * 57 = 355$ cycles in actor b. Similarly, *os_set_scenario* iterates over all outgoing channels. Here we measured a static component of 142 plus 38 for a self-edge, 56 for a non-self FIFO, and 45 per token. Calling this function in actor b costs $142+38+2*56+8*45 = 652$ cycles. Altogether the execution time of the system calls depends on the scenario size and the number of consuming and producing FIFOs that an actor has, and can easily be calculated.

To show the benefit of our approach, we processed the example FSM-SADF application with the SDF^3 toolflow. First we annotated the graph with artificial execution times. All actors are assigned an execution time of $1 \cdot 10^5$ cycles, except c and d which both get $15 \cdot 10^5$ cycles and f which gets $30 \cdot 10^5$. We also created a synchronous dataflow (SDF) application that captures both scenarios at the same time, i.e. it has a graph that resembles S_2 in Figure 1 but f is replaced by bf and g by cdg. The execution time of bf is equal to $\max(\tau_b, \tau_f)$ with τ_i the execution time of an actor, and that of cdg to $\max(\tau_c + \tau_d, \tau_g)$. The values of τ_c and τ_d are added because these are not separate actors anymore and cannot be mapped on separate resources. We compensated for that by selecting an FSM-SADF mapping in which these actors are also on the same core. The throughput constraint was set to $2 \cdot 10^{-10}$, which means an iteration must be completed every $5 \cdot 10^9$ cycles. There are 10 TDM slots with a length of $4 \cdot 10^5$ cycles each. For both applications, the mapping resulting in the lowest number of TDM slots was selected. The results are shown in the first two rows of Table II. The number of TDM slots is listed as $c_1 + c_2$. We see that the throughput constraint can be satisfied with 7 slots using FSM-SADF versus 9 slots with SDF. The advantage in TDM slots cannot be used to quantify our approach directly because the application is artificial, but we argue that the setup reflects realistic real-time (streaming)

applications. We conclude that the FSM-SADF programming model can offer a tight bound on throughput that enables to save resources compared to SDF, but the actual benefit depends on the application.

As for the memory, we compared the two setups in terms of instruction and data memory. Channels are are placed in communication memories if they connect actors mapped to different cores, and are allocated on the heap otherwise. Therefore we added the usage of communication memories and heap to the data memory. Looking again at the first two rows of Table II, we conclude that the implementation of this simple example has a cost of 1032 bytes of instruction memory and 215 bytes of data memory. The modifications to the POSE OS furthermore add 900 bytes of instruction memory per core (not shown in the table). The relative increase in the memory budget is meaningless as the actors are empty, but the absolute cost gives a realistic indication of the cost for a graph of this size. Overall, the run-time costs are minimal even for our trivial example. The predictable computation cost added to each actor is easily nullified by the $8 \cdot 10^5$ cycles that are gained. The increase in memory and communication budgets will be negligible for realistic applications.

To compare the implementation with alternative A described in Section V, we also used SDF^3 to obtain TDM slot allocations for the individual scenarios. Memory budgets are not exact but extrapolated from the second setup described at the start of this section. The switching cost was modelled by multiplying the combined size of data and instruction memory with 21 cycles as described in Section IV-B, and included in scenario S_0. The results can be found in the last four rows of Table II. We see that S_0 needs two slots to execute and load a scenario, and that the application uses as many slots as the SDF solution. In other words, the benefit of the tighter bound throughput is negated by the cost for switching. The total memory budget is lower than that of the other two, although implementation overhead is not included. If the size of the application increases, the cost for switching will rapidly increase and affect the throughput. Based on these numbers we conclude that this alternative is indeed only relevant if the memory budget is leading, or if the bundle loading time can be reduced significantly

VIII. Conclusion

The FSM-SADF model of computation can account for dynamic variations in an application, which results in a tighter throughput bound compared to SDF while it is still analysable. In this work we propose a programming model that allows intuitive implementation of FSM-SADF programs. To efficiently switch scenarios, we argue that the scenario graphs and FSM should be intertwined. Implementation tradeoffs that we identified concern scheduling, resource budgets, scenario switch time, scenario pipelining, and effort for run-time modifications. We explored three implementation alternatives and concluded that encapsulating all scenarios in one VP is the most beneficial choice on this platform. It features minimal switching time, allows pipelining, and is compatible with other dataflow types. To avoid variable rates and changing the SO schedule dynamically we proposed to fire all actors each iteration. Experiments confirm that there is no switching time, but a predictable run-time cost is added to the wrapper of each actor. The increase in the memory and communication budgets is negligible for realistic applications. We show that the tight bound on throughput offered by FSM-SADF reduces the number of required TDM slots and thus the overall resource budget. A model of the 1^{st} alternative shows that it has no througput benefit compared to SDF, but the memory budget is lower.

Acknowledgments

This work was partially funded by projects CATRENE CA505 BENEFIC, CA703 OpenES, CT217 RESIST; ARTEMIS 621429 EMC2 and 621353 DEWI.

References

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

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

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

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

[5] F. Siyoum, "Worst-case temporal analysis of dynamic streaming applications," Ph.D. dissertation, TUE, 2014.

[6] S. Stuijk et al., "SDF^3: SDF For Free," in *ACSD*, 2006.

[7] H. Kopetz, *Real-Time Systems*. Springer, 2011.

[8] T. A. Henzinger et al., "Giotto: a time-triggered language for embedded programming," *IEEE Proceedings*, 2003.

[9] E. Lee and D. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions*, 1987.

[10] J. Eker and J. W. Janneck, "Cal language report: Specification of the cal actor language," UC Berkeley, Tech. Rep., 2003.

[11] M. Geilen et al., "Predictable dynamic embedded data processing," in *SAMOS*, July 2012.

[12] H. Kopetz et al., "Composability in the time-triggered system-on-chip architecture," in *SOC*, 2008.

[13] D. Broman et al., "Precision timed infrastructure: Design challenges," in *ESLsyn*, 2013.

[14] T. Ungerer et al., "parMERASA - Multi-core Execution of Parallelised Hard Real-Time Applications Supporting Analysability," in *DSD*, 2013.

[15] Martin Schoeberl et al., "T-CREST: time-predictable multi-core architecture for embedded systems," *Elsevier Journal on Sys. Arch.*, 2015.

[16] S. Trujillo et al., "MultiPARTES: multi-core partitioning and virtualization for easing the certification of mixed-criticality systems," *Microprocessors and Microsystems*, 2014.

[17] K. Goossens et al., "Æthereal Network on Chip: Concepts, Architectures, and Implementations," *IEEE Des. Test*, 2005.

[18] A. Nelson et al., "CoMik: A Predictable and Cycle-Accurately Composable Real-Time Microkernel," in *DATE*, 2014.

[19] S. Sinha et al., "Composable and predictable dynamic loading for time-critical partitioned systems," in *DSD*, 2014.

[20] A. Hansson et al., "Design and implementation of an operating system for composable processor sharing," *Microproc. and Microsystems*, 2011.

[21] A. B. Nejad et al., "A Unified Execution Model for Multiple Computation Models of Streaming Applications on a Composable MPSoC," in *Elsevier Journal of Systems Architecture*, 2013.

[22] S. Goossens et al., "The CompSOC design flow for virtual execution platforms," in *FPGAworld*. ACM, 2013.