Self-Powering Dataflow Networks – Concepts and Implementation

Abrarul Karim, Joachim Falk, Dennis Schmidt, and Jürgen Teich
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)
Erlangen, Germany; *firstname.lastname*@fau.de

***Abstract*—Dataflow networks play a vital role in modeling and analyzing stream-processing systems in an analytic way, including digital signal and image processing systems. In this paper, we first present a system-level approach to synthesize such dataflow networks automatically to systems of communicating hardware actors connected by FIFO buffers. Although such data-triggered networks of (internally clocked) actors can achieve very high throughputs, the potential to power actors down in times of unavailability of data has not been addressed so far in any research. Here, we show that by refinement of the firing state machine of each actor in a given network, we enable the design of *self-powering dataflow networks* while exploiting either clock gating or power gating as a means to save power in times of inactivity of each individual actor in a network. The gains of self-powering dataflow networks in terms of power and energy savings when powering down and up actors dynamically is shown for different data arrival patterns and rates in detailed experiments for multiple IoT system applications. These systems are often working in normally-off mode and woken up only upon the availability of data. For these, drastic energy savings are reported.**

***Index Terms*—dataflow networks, self-powering systems**

I. Introduction

Stream processing is used in many computationally challenging applications such as digital signal and image processing, cryptography, and many others. Related applications can be effectively modeled using *Dataflow Graphs (DFGs)*, where each node represents a computational kernel called an *actor* that is evoked to process data on incoming edges to create data on output edges depending on the satisfaction of certain *firing rules*. Dataflow networks naturally enable the parallel execution of multiple actors, if enabled to fire. As an example, Fig. 1 shows a Sobel filter application modeled by a DFG.

Whereas formal techniques for model-based performance analysis of different types of DFGs have existed for a long time, e.g., [1]–[3], only a few approaches and tools are available to synthesize such DFGs directly to hardware, e.g., [4]–[6]. Such hardware realized DFGs could be particularly beneficial for lightweight Internet of Things (IoT) applications, often facing stringent energy constraints, particularly when operating on batteries and using energy harvesting. Still, many remotely deployed devices may require a parallel hardware implementation to perform real-time signal analysis, filtering, and compression in order to save energy for only transmitting filtered result data rather than raw streams of, e.g., audio, image, and other sources of sensor data. A need and trend can thus be observed to provide in situ also highly computationally intensive tasks like image/video processing (e.g., compression), machine learning (e.g., image recognition), etc.

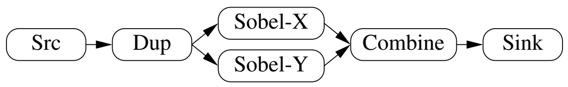


Figure 1: Sobel filter image processing dataflow graph. Communication between actors (vertices) is realized via FIFO channels (directed edges).

This work has been partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under project grant 530178246.

At the same time, highly energy-efficient implementations are required, which may seem contradictory. As major contributions of this paper, we propose (i) a systematic system-level approach to synthesize a given dataflow network directly in hardware. To save energy for such hardware networks when actors are inactive, we then introduce (ii) the concept of *self-powering dataflow networks*. Such networks can conserve energy by self-powering down each individual actor upon unavailability of incoming data and powering up again only upon data arrival. To realize such a self-powering behavior, each actor's *firing state machine*, which implements its firing rules and, thus, communication behavior, is automatically transformed into a firing state machine augmented with *power management transitions*. Presented is a Finite-State Machine refinement algorithm to move an actor into a *sleep state* if, after a current firing, a next firing is not possible, e.g., due to the lack of incoming data. (iii) The synthesis flow can be extended to support multiple power-saving techniques. In this paper, we emphasize on *clock gating* (to save dynamic power), but extensions to also support *power gating* to reduce actor idle power are mentioned as well. The latter also requires extracting and retaining the internal actor state to avoid losing important state information. Finally, (iv) we provide detailed experimental results for several applications from signal processing to analyze achievable power and energy savings in dependence of different distributions of input data.

To the best of our knowledge, this is the first work on *self-powering dataflow networks* in terms of concept, automatic model-based synthesis flow, and analysis.

Section II defines fundamentals. Section III outlines our synthesis flow targeting Application Specific Integrated Circuits (ASICs). Section IV introduces self-powering dataflow networks and formal transformations of each actor's firing state machine to support different power-saving techniques, including clock and power gating. Section V presents the obtained energy savings, and Section VI concludes the paper.

II. Fundamentals and Notations

Actor networks [1]–[3] can be described by graphs with nodes corresponding to actors exchanging data over edges. By annotating such nodes and edges with additional information such as execution times, number of input data elements needed to fire, memory requirements, etc., a formal analysis of a network in terms of performance-relevant information, e.g., throughput, schedulability, or memory requirements, gets possible. However, in order to provide a system-level synthesis flow from actor network to direct hardware implementation, also the behavior of each actor needs to be modeled properly using either a normal programming language or a domain-specific language such as CAL [7].

In this paper, we consider actor networks described in the SysteMoC [8] language. SyteMoC is based on the formal dataflow model introduced in [9] and is realized as a C++ class library on top of *SystemC* [10]. In the following, we introduce the terms of a *network graph* composed of SysteMoC actors and First In First Out (FIFO) channels, e.g., as shown in Fig. 2.

Actors communicate with each other by means of (data) *tokens* that are transmitted over the FIFO channels. Tokens already present before the first actor firing are called *initial tokens*, which are represented by black dots on the edges of the network graph (cf. channel c_3 in Fig. 2). More formally, a network graph is defined as follows:

Definition 2.1 (Network Graph [8]): A *network graph* is a directed bipartite graph $\mathbf{g} = (V, E)$ containing a set of vertices $V = C \cup A$ partitioned into *channels* C and *actors* A, a set of directed edges $e = (v_{\text{src}}, v_{\text{snk}}) \in E \subseteq (C \times A.I) \cup (A.O \times C)$ from channels $c \in C$ to actor input ports $i \in A.I$ as well as from actor output ports $o \in A.O$ to channels. These edges are further constrained such that exactly one edge is incident to each actor port and the in-degree and out-degree of each channel in the graph is exactly one. Finally, there are the *delay* function $\delta : C \to \mathbb{N}_0$, *capacity* function $\gamma : C \to \mathbb{N}$, and *size* function $\varphi : C \to \mathbb{N}$ that assign each channel a number of initial tokens, a maximal number of tokens that can be stored, and the token size in bits, respectively.

A SysteMoC actor (cf. Fig. 3) is defined as follows:

Definition 2.2 (Actor [8]): An actor is a tuple $a = (I, O, \mathcal{F}, \mathcal{R})$ containing a set of *actor input ports* I and *actor output ports* O, the *actor functionality* $\mathcal{F} = \mathcal{F}_{\text{action}} \cup \mathcal{F}_{\text{guard}}$ partitioned into a set of *actions* and a set of *guard functions*, as well as the *actor Finite-State Machine (FSM)* $\mathcal{R}$. For completeness, there is a notion of a *functionality state* $q_{\text{func}} \in Q_{\text{func}}$ of an actor, which denotes its internal state, given implicitly by the set of actor-internal C++ variable declarations.[1]

Moreover, firing rules controlling token consumption and production are realized by an *actor FSM* as given below:

Definition 2.3 (Actor FSM [8]): The FSM $\mathcal{R}$ of an actor a is a tuple (Q, q_0, T) containing a finite set of *states* Q, an *initial state* $q_0 \in Q$, and a finite set of *transitions* T. A *transition* $t \in T$ itself is a tuple $(q_{\text{src}}, k, f_{\text{action}}, q_{\text{dst}})$ containing the source state $q_{\text{src}} \in Q$, from where the transition is originating,

[1]This state needs to be extracted and retained during power gating in order not to loose important information when powering down an actor.

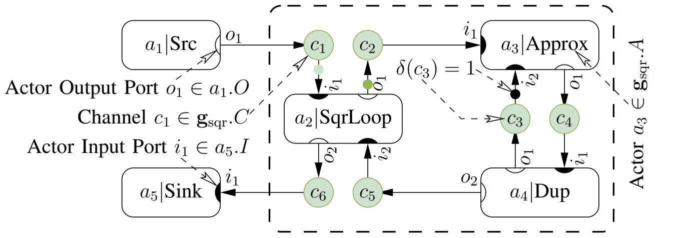
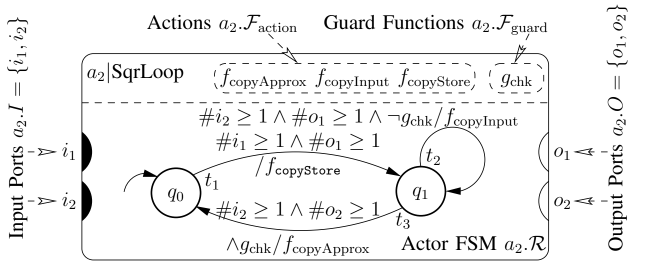


Figure 2: *Network graph* $\mathbf{g}_{\text{sqr}}$ modeling Newton's iterative algorithm for calculating the square roots of an infinite input sequence of numbers generated by the `Src` actor a_1. The square root values are generated by the `SqrLoop` actor a_2, which triggers approximation steps via the actors a_3 and a_4 until an error bound is satisfied. Then, the result is transported to the `Sink` actor a_5. Tokens are represented by the dots on the edges between channels and actors. The dashed box indicates the subgraph to be synthesized in hardware.



Actions $a_2.\mathcal{F}_{\text{action}}$
Guard Functions $a_2.\mathcal{F}_{\text{guard}}$
a_2|SqrLoop
$f_{\text{copyApprox}}$ $f_{\text{copyInput}}$ $f_{\text{copyStore}}$
g_{chk}
Input Ports $a_2.I = \{i_1, i_2\}$
i_1
i_2
$\#i_2 \geq 1 \land \#o_1 \geq 1 \land \neg g_{\text{chk}} / f_{\text{copyInput}}$
$\#i_1 \geq 1 \land \#o_1 \geq 1$
$/f_{\texttt{copyStore}}$
t_1
t_2
q_0
q_1
$\#i_2 \geq 1 \land \#o_2 \geq 1$
t_3
$\land g_{\text{chk}} / f_{\text{copyApprox}}$
Actor FSM $a_2.\mathcal{R}$
o_1
o_2
Output Ports $a_2.O = \{o_1, o_2\}$

Figure 3: Shown is the `SqrLoop` actor a_2 from the network graph $\mathbf{g}_{\text{sqr}}$. This actor is composed of *input ports* I and *output ports* O, its *functionality* $\mathcal{F}_{\text{func}}$, and the *actor FSM* $\mathcal{R}$ determining the actor's communication behavior. The *input guard* $\#i_x \geq n$ and the *output guard* $\#o_y \geq m$ on a *transition*, respectively, test the availability of at least n tokens on the actor input port i_x and at least m free places on the actor output port o_y.

and the destination state $q_{\text{dst}} \in Q$, which will become the next current state after the execution of the transition starting from the current state q_{src}. Furthermore, if the transition t is taken, then an action f_{action} from the set of functions of the *actor functionality* $a.\mathcal{F}_{\text{action}}$ will be computed. Finally, the execution of a transition t itself can be guarded by a *transition guard* k.

An *actor firing* involves the consumption of tokens from its *input FIFOs*, the computation of the actor functionality on this data, and finally, the production of tokens on its *output FIFOs*. Thus, an actor can fire if all its input FIFOs have at least as many tokens available as the actor firing would consume, and its output FIFOs have sufficient space to fit the tokens the firing would produce. In our running example, assume that one token is present in channel c_1 (e.g., the light green dot shown in Fig. 2) and channel c_2 is empty. Then, transition $t_1 : \#i_1 \geq 1 \land \#o_1 \geq 1 / f_{\texttt{copyStore}}$ of actor a_2 (cf. Fig. 3) can fire, as at least one token is present in channel c_1 connected to input port i_1 and at least one free place is available in channel c_2 connected to output port o_2. After actor a_2 has fired, channel c_1 will be empty, and channel c_2 will contain the produced token (e.g., the dark green dot shown in Fig. 2).

III. Automatic Hardware Synthesis

As our first contribution, an approach for the fully automatic hardware synthesis of previously defined actor networks is presented. Section III-A introduces how a netlist of communicating hardware actors and channels is generated automatically from a SysteMoC actor network, while Section III-B elaborates on the hardware realization of a SysteMoC channel.

A. Synthesis Framework

In order to exploit modern behavioral (high-level) synthesis to bridge the abstraction layers from system level to logic level, a SysteMoC description is first converted into a synthesizable SystemC description. This transformation of SysteMoC actors to SystemC is performed by a source-to-source compiler based on clang [11]. The transformed SystemC description can then be used for simulation and automatic behavioral synthesis using High-Level Synthesis (HLS) tools. In our approach, a Verilog description for each SysteMoC FIFO channel is generated from the template described in Section III-B. Register Transfer Level (RTL) synthesis tools then translate the actor and FIFO modules to gate-level netlists. Moreover, a top-level netlist is generated, which instantiates and connects the actor and channel netlists, according to the initial SysteMoC graph description. This netlist can then be directly passed to ASIC place and route tools. Alg. 1 presents a pseudo code summarizing the above-mentioned steps.

Algorithm 1: Netlist Generation

```
   Input: A SysteMoC network graph $\mathbf{g}$
   Output: A netlist ready for ASIC place and route
1  $A''' \leftarrow \emptyset$; $C''' \leftarrow \emptyset$;
2  foreach SysMoC_actor $a \in \mathbf{g}.A$ do
3      SystemC_actor $a' =$ convert(SysMoC_actor $a$);
4      Verilog_actor $a'' =$ HLS_synthesis(SystemC_actor $a'$);
5      Netlist_actor $a''' =$ RTL_synthesis(Verilog_actor $a''$);
6      $A''' \leftarrow A''' \cup \{a'''\}$;
7  foreach SysMoC_channel $c \in \mathbf{g}.C$ do
8      Verilog_channel $c'' =$
         Fill_channel_template(SysMoC_channel $c$, channel_template);
9      Netlist_channels $c''' =$ RTL_synthesis(Verilog_channel $c''$);
10     $C''' \leftarrow C''' \cup \{c'''\}$;
11 return generate_netlist($\mathbf{g}, A''', C'''$);
```

B. SysteMoC Channel Realization

A SysteMoC channel is realized in hardware by a ring buffer that has an independent read (shaded light red) and an independent write interface (shaded light blue in Fig. 4). As complex guard conditions may involve testing any token on a channel for presence and/or value, each memory element of a channel must be accessible independently like a random access memory (even if tokens are consumed only in FIFO order according to dataflow semantics). Thus, a channel must provide a read and a write interface for access to so-called *random access regions* that are addressed relative to a *read pointer* (`rp`) and, respectively, *write pointer* (`wp`) via `rel_rd_addr` and `rel_wr_addr` signals.

In Fig. 4, an example configuration is shown that could be the implementation of the channel c_2 in Fig. 2, with the output port o_1 of a_2 connected to the write interface of the channel (represented by the signals on the left) and the read interface connected to the input port i_1 of a_3 (represented by the signals on the right). In this example configuration, the random access regions comprise two tokens (i.e., the two tokens highlighted in light red), respectively three free

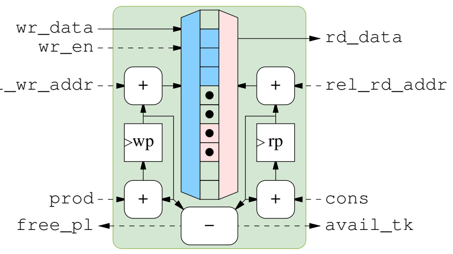


Figure 4: RTL description of a SysteMoC channel and the provided interface. Solid lines represent data signals, while dashed lines represent control signals.

places (i.e., the three memory positions highlighted in blue).[2] A channel needs to support three basic operations in total. These include a non-destructive read as well as a destructive read (consume) and a destructive write (produce). Data is transmitted via the `rd_data` and `wr_data` signals, where `wr_en = 1` distinguishes a write operation from a read operation. Moreover, destructive reads/writes are selected by asserting a corresponding binary signal `cons`, respectively `prod`. In that case, the FIFO advances the value of a write pointer register (`wp`), respectively read pointer register (`rp`) modulo the token capacity of the FIFO.

Finally, the signals `avail_tk` and `free_pl` indicate how many tokens or free places are available in the channel.

To instantiate a channel in hardware, a *channel template* has been defined with the following parameters: *token size*, denoted by $\varphi(c)$; *channel capacity* (size of channel memory in terms of number of tokens that can be stored), denoted by $\gamma(c)$; *read random access region size*, e.g., two in the given example, and *write random access region size*, e.g., three in the given example. Figure 4 shows a generic schematic of a SysteMoC channel hardware template.

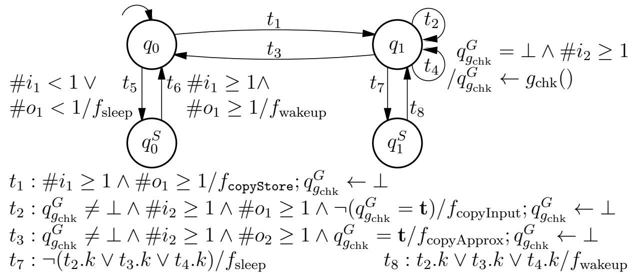
IV. Self-Powering Dataflow Networks

Under the term *self-powering dataflow networks*, we want to exploit now the opportunity for power savings of actor networks by only powering each actor once its fireability gets established, e.g., by arrival on input data or occurence of free spaces at output channels and powering it down otherwise. We show that methodologically, this can be achieved by an actor transformation in two steps. First, the actor FSM $\mathcal{R}$ of each actor is separated from its actor functionality $\mathcal{F}$, as the FSM controller must always be powered on. Second, the actor FSM is augmented by additional states and transitions as described in Section IV-A to power up, respectively down the hardware implementing the actor functionality $\mathcal{F}$. As technical means of power management, we show that the transformation presented next directly enables the application of well-known techniques for power management, namely *clock gating* to save dynamic power (Section IV-B) and *power gating* to save also static power (Section IV-C).

[2]The size of the write access region is solely determined by the maximum number of tokens that any transition of the actor FSM that fires into this FIFO channel produces. Moreover, the size of the read access region is solely determined by the maximum number of tokens that the FSM of the actor that reads tokens from the channel consumes, see [12] for more details.

A. Power Controller Generation for each Actor

To generate a power controller for each actor, we propose a transformation of its firing FSM using Alg. 2 such that clock gating or power gating techniques are systematically exploited to save power in times of inactivity. Through this FSM transformation, we enable the synthesis of self-powering dataflow networks. As an example, Fig. 5 shows the result of this transformation when applied to the firing FSM of actor `SqrLoop` depicted in Fig. 3.



q_0
q_1
q_0^S
q_1^S
t_1
t_2
t_3
t_4
$q^G_{g_{\text{chk}}} = \bot \wedge \#i_2 \geq 1$
$/q^G_{g_{\text{chk}}} \leftarrow g_{\text{chk}}()$
$\#i_1 < 1 \vee$
$\#o_1 < 1/f_{\text{sleep}}$
t_5
t_6
$\#i_1 \geq 1 \wedge$
$\#o_1 \geq 1/f_{\text{wakeup}}$
t_7
t_8
$t_1 : \#i_1 \geq 1 \wedge \#o_1 \geq 1/f_{\texttt{copyStore}}; q^G_{g_{\text{chk}}} \leftarrow \bot$
$t_2 : q^G_{g_{\text{chk}}} \neq \bot \wedge \#i_2 \geq 1 \wedge \#o_1 \geq 1 \wedge \neg(q^G_{g_{\text{chk}}} = \mathbf{t})/f_{\text{copyInput}}; q^G_{g_{\text{chk}}} \leftarrow \bot$
$t_3 : q^G_{g_{\text{chk}}} \neq \bot \wedge \#i_2 \geq 1 \wedge \#o_2 \geq 1 \wedge q^G_{g_{\text{chk}}} = \mathbf{t}/f_{\text{copyApprox}}; q^G_{g_{\text{chk}}} \leftarrow \bot$
$t_7 : \neg(t_2.k \vee t_3.k \vee t_4.k)/f_{\text{sleep}}$
$t_8 : t_2.k \vee t_3.k \vee t_4.k/f_{\text{wakeup}}$

Figure 5: Transformed firing FSM resulting from applying Alg. 2 to the firing FSM of the `SqrLoop` actor depicted in Fig. 3 realizing dynamic power management for this actor via the f_{sleep} and f_{wakeup} actions entering, respectively, leaving the sleep states q_0^S and q_1^S. States q_0 and q_1 as well as transitions t_1 to t_3 are derived from the original actor FSM. Transition t_4 is added to evaluate the guard g_{chk}, while transitions t_5 to t_8 are included for power management.

Alg. 2 receives the actor FSM $a.\mathcal{R}$ of an actor a as its input and returns an augmented FSM $\mathcal{R}^S$ realizing power management and actor firing control (see Ln. 1). First, in Ln. 2, the augmented FSM starts with an empty set of transitions T that the algorithm will later populate. In contrast, the state space Q and the initial state q_0 of the actor FSM are copied as a starting point for the augmented FSM in Lns. 3 to 4. In Fig. 5, this copying results in the states q_0 and q_1. Subsequently,

Algorithm 2: FSM transformation

```
1   Function transformFSM($\mathcal{R}$)
        Input  : Actor FSM $\mathcal{R}$ of an actor $a$
        Output: Augmented FSM $\mathcal{R}^S$ realizing power management
2       $T \leftarrow \emptyset$    // Start with empty transition set
3       $Q \leftarrow \mathcal{R}.Q$    // Copy states from FSM $\mathcal{R}$
4       $q_0 \leftarrow \mathcal{R}.q_0$    // Copy initial state of FSM $\mathcal{R}$
5       $\mathbf{Q}^G \leftarrow \{\bot, \mathbf{f}, \mathbf{t}\}^{|\mathcal{F}_{\text{guard}}|}$    // Guard state space
6       $\mathbf{q}_0^G \leftarrow \bot^{|\mathcal{F}_{\text{guard}}|}$    // Initial guard state
        /* For all states of $\mathcal{R}$ */
7       foreach $q \in \mathcal{R}.Q$ do
            /* For all guards utilized by state $q$ */
8           foreach $g \in \mathcal{F}_{\text{guard}}(q)$ do
9               $k \leftarrow (q_g^G = \bot) \wedge (\bigvee_{t \in \mathcal{R}.T(q) \wedge g \in \mathcal{F}_{\text{guard}}(t.k)} t.k^{\text{in}})$
10              $T \leftarrow T \cup \{(q, k, q_g^G \leftarrow g(), q)\}$
            /* For all transitions leaving state $q$ */
11          foreach $t \in \mathcal{R}.T(q)$ do
12              $k \leftarrow (\bigwedge_{g \in \mathcal{F}_{\text{guard}}(t.k)} q_g^G \neq \bot) \wedge replace(t.k)$
13              $T \leftarrow T \cup \{(t.q_{\text{src}}, k, t.f_{\text{action}}(); \mathbf{q}^G \leftarrow \mathbf{q}_0^G, t.q_{\text{dst}})\}$
14          $Q \leftarrow Q \cup \{q^S\}$    // Add sleep state $q^S$
15          $k \leftarrow \bigvee_{t \in T(q)} t.k$    // Action/guard eval. possible?
16          $T \leftarrow T \cup \{(q, \neg k, f_{\text{sleep}}, q^S)\}$    // Sleep trans.
17          $T \leftarrow T \cup \{(q^S, k, f_{\text{wakeup}}, q)\}$    // Wakeup trans.
18      $\mathcal{R}^S \leftarrow (Q, q_0, \mathbf{Q}^G, \mathbf{q}_0^G, T)$
19      return $\mathcal{R}^S$
```

Lns. 5 to 6 define a *guard state space* $\mathbf{Q}^G$ with an initial guard state $\mathbf{q}_0^G = \bot^{|\mathcal{F}_{\text{guard}}|}$ that denotes that each guard's initial *guard evaluation state* is unevaluated ($\bot$). The transformation is performed by the loop in Lns. 7 to 17 that iterates over each state $q \in \mathcal{R}.Q$ of the initial actor FSM $\mathcal{R}$, while Ln. 18 assembles the augmented FSM to return it in Ln. 19.

In the loop body, the algorithm iterates (Lns. 8 to 10) over all guard functions $g \in \mathcal{F}_{\text{guard}}(q)$ used by a state q, e.g., for the actor FSM depicted in Fig. 3, $\mathcal{F}_{\text{guard}}(q_0) = \emptyset$ but $\mathcal{F}_{\text{guard}}(q_1) = \{g_{\text{chk}}\}$.[3] For each such guard function g, a self-loop transition is added to state q (Ln. 10) to evaluate it, updating its guard evaluation state q_g^G, e.g., as shown in Fig. 5 by the self-loop transition t_4 for the state q_1 evaluating the guard function g_{chk} updating its evaluation state $q^G_{g_{\text{chk}}}$ from unknown ($\bot$) to either true ($\mathbf{t}$) or false ($\mathbf{f}$). For this purpose, Ln. 9 defines a transition guard k that checks that the guard g is currently unevaluated ($q_g^G = \bot$) and that there are sufficient available input tokens to evaluate the guard, e.g., to evaluate the guard function g_{chk} at least one input token is required on input port i_2, resulting in the transition t_4, as depicted in Fig. 5.

Subsequently, in Lns. 11 to 13, the algorithm iterates over all transitions $t \in \mathcal{R}.T(q)$ of the actor FSM $\mathcal{R}$ leaving state q. Next, the transition guard $t.k$ is modified in Ln. 12 such that guard functions are not used directly, but instead, their guard evaluation state is checked to allow the augmented FSM to still operate when the actor functionality $\mathcal{F}$, which contains the guard functions, is powered down. For example, the transition guard $\#i_2 \geq 1 \wedge \#o_2 \geq 1 \wedge g_{\text{chk}}$ of the transition t_3 from the actor FSM shown in Fig. 3 is modified by adding a check that the guard evaluation state of the utilized guard function g_{chk} is evaluated, i.e., $q^G_{g_{\text{chk}}} \neq \bot$, and the guard function is replaced, e.g., *replace*$(\#i_2 \geq 1 \wedge \#o_2 \geq 1 \wedge g_{\text{chk}}) = \#i_2 \geq 1 \wedge \#o_2 \geq 1 \wedge q^G_{g_{\text{chk}}} = \mathbf{t}$. Moreover, the transition action $t.f_{\text{action}}$ is extended to reset the guard evaluation state of all guards ($\mathbf{q}^G$) to unevaluated ($\mathbf{q}_0^G$), and the modified transition is added to the augmented FSM in Ln. 13. The resulting modified transitions t_1 to t_3 are shown in Fig. 5.

Finally, a sleep state q^S for each actor FSM state q is added to the state set Q in Ln. 14, and the transition set T is extended in Lns. 16 and 17 with corresponding sleep and wakeup transitions triggering the actions f_{sleep}, respectively, f_{wakeup}. To determine if the sleep transition can be taken, Ln. 15 defines a condition k that checks if any other transition leaving state q can be taken, i.e., an actor action can be executed, or a guard can be evaluated. If not ($\neg k$, cf. Ln. 16), the sleep transition is enabled to enter the sleep state q^S. Conversely, if condition k becomes true again, e.g., because additional input tokens have arrived at the actor inputs or, alternatively, free space became available at the actor outputs, the sleep state q^S is left via the wakeup transition, see Ln. 17. In our example, this results in the transitions t_5 to t_8 shown in Fig. 5.

[3]More formally, we will use the notation $\mathcal{F}_{\text{guard}}(k)$ to denote the set of guard functions that is used by a transition guard k, e.g., $\mathcal{F}_{\text{guard}}(\#i_1 \geq 1 \wedge (g_1 \vee \neg g_2)) = \{g_1, g_2\}$, and $\mathcal{F}_{\text{guard}}(q)$ to denote the set of guard functions where each guard is used by at least one transition outgoing from state q, i.e., $\mathcal{F}_{\text{guard}}(q) = \cup_{t \in T(q)} \mathcal{F}_{\text{guard}}(t.k)$ where the notation $T(q)$ denotes the set of all transitions leaving state q, i.e., $T(q) = \{t \in T \mid t.q_{\text{src}} = q\}$.

B. Clock Gating

In order to exploit clock gating, only the two actions sleep and wakeup (f_{sleep} and f_{wakeup}) as described in Section IV-A need to be implemented as depicted in Fig. 6, showing the augmented FSM $\mathcal{R}^S$ and the clocked actor functionality $\mathcal{F}$. To take a transition to sleep mode, only the `clk_en` signal is set to low. Once data is available again, the wakeup transition is taken, and the `clk_en` signal is set to high, restoring regular actor functionality.

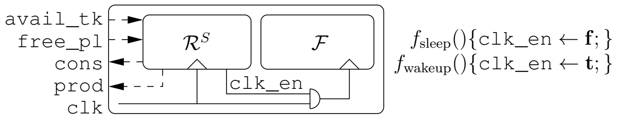

avail_tk
free_pl
cons
prod
clk
$\mathcal{R}^S$
$\mathcal{F}$
clk_en
$f_{\text{sleep}}()\{\texttt{clk\_en} \leftarrow \mathbf{f};\}$
$f_{\text{wakeup}}()\{\texttt{clk\_en} \leftarrow \mathbf{t};\}$

Figure 6: Schematic of a clock-gated actor. The augmented FSM $\mathcal{R}^S$ is clocked as usual. The actions f_{sleep} and f_{wakeup} change an internal clock enable signal `clk_en`.

C. Power Gating

Power gating techniques can be exploited on top to reduce leakage power. To power down an actor without any data loss, the internal states must be saved as shown in Fig. 7. Only then can the augmented FSM $\mathcal{R}^S$ power down the actor functionality $\mathcal{F}$. To wake up, the actor functionality is powered up, and data is restored using retention registers that are usually concatenated to form a scan chain implementation and restored serially, as shown in Fig. 7.

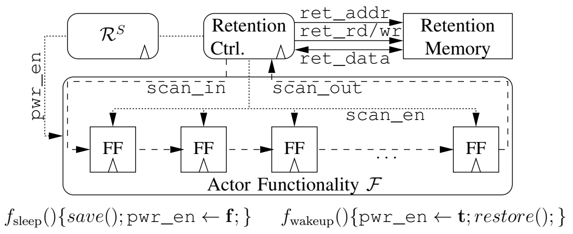

$\mathcal{R}^S$
Retention Ctrl.
ret_addr
ret_rd/wr
ret_data
Retention Memory
pwr_en
scan_in
scan_out
scan_en
FF
FF
FF
FF
...
Actor Functionality $\mathcal{F}$
$f_{\text{sleep}}()\{save();\texttt{pwr\_en} \leftarrow \mathbf{f};\}$
$f_{\text{wakeup}}()\{\texttt{pwr\_en} \leftarrow \mathbf{t};restore();\}$

Figure 7: Schematic showing an augmented FSM $\mathcal{R}^S$ that triggers a *retention controller* to save and restore the actor functionality state during a power down and up phase in a *retention memory*. Control signals are shown as dotted lines, and dash lines represent the *scan chain* accessing all Flip Flops (FFs) of the synthesized actor functionality. The retention memory may be realized using emerging Non-Volatile Memory (NVM) technologies.

V. Experimental Setup and Results

In this section, we provide detailed experiments for multiple applications to analyze the potential of self-powering dataflow networks in terms of *power savings* and *achievable throughputs*. For this purpose, we introduce the notions of *utilization* and *intermittency*, defining different *input activation patterns* used for power analysis. We then shortly introduce three considered applications and their actor networks and functions. Subsequently, we apply different input stimuli to these applications to analyze clock gating in terms of energy savings and potential changes in achievable throughput.

A. Introduction of Utilization and Intermittency

Our following test applications are all stimulated by N activations over a time period of T clock cycles. Let the *data initiation interval* of a given application be denoted by d_{ii} (in clock cycles). In order to be able to analyze power savings in times of inactivity, we define a parameter U as *utilization*. For a given value of $U \in]0, 100]\,\%$, T is then given by $T(U) = \frac{d_{\text{ii}} \cdot N}{U/100}$ (clock cycles). Moreover, stimuli can arrive either in bursts or intermittently. To reflect this, we introduce a second parameter $I \in [0, 100]\,\%$ called *intermittency* that controls the number $B = \lfloor (N-1) \cdot I/100 + 1 \rfloor$ of bursts in which $\frac{N}{B}$ activations are stimulated as soon as possible.

E.g., let $N = 4$ and $d_{\text{ii}} = 4$ with utilization values $U = 100\,\%$ and $U = 20\,\%$, resulting in $T(100) = \frac{4 \cdot 4}{1.00} = 16$ and $T(20) = \frac{4 \cdot 4}{0.20} = 80$ clock cycles. For the case of $U = 20\,\%$ as well as chosen intermittency values of $I = 0\,\%$, $I = 50\,\%$, and $I = 100\,\%$, respectively, 1, 2, and 4 bursts result. Figure 8 shows the corresponding input activations.

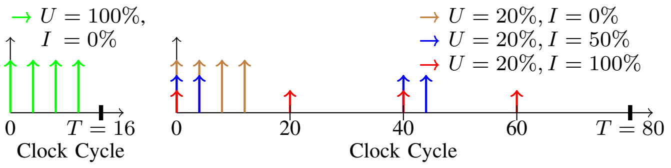


Figure 8: Input activation patterns for different values of U and I.

B. Considered Applications and Experimental Setup

The test applications considered in the following include: a) an FIR-Filter `FIR`, b) a video processing application `VID` according to Fig. 1, and c) the `SqrRoot` actor network application as introduced as the running example in Section II.

For all three test applications, we first generate the hardware netlist starting with a SysteMoC description using the synthesis flow described in Section III-A, i.e., Alg. 1. Then, after applying the transformation of the actor FSM $\mathcal{R}$ of each actor as described in Section IV-A, we synthesize the self-powering actor network circuit such to implement clock gating by implementing the corresponding actions for powering down and up of each actor as described in Section IV-B. Finally, the resulting netlist is simulated over the time period $T(U)$ using stimuli generated in a testbench with given values of U and I, and the average power and energy is determined using Questa Advanced Simulator [13] and Synopsys PrimePower [14].

E.g., for the `FIR` design ($d_{\text{ii}} = 4$) employing clock gating stimulated with $U = 20\,\%$ and $I = 100\,\%$, Fig. 9 depicts the resulting power and energy consumption stemming from the shown sleep and wakeup behavior. I.e., the actor goes to sleep mode once the processing of an activation is complete. Thereby, the power consumption is drastically reduced. Only once the next activation arrives, as indicated by the red arrows, the actor is powered up, and the processing continues. In our implementation, a sleep-wakeup cycle has an overhead of just 3 clock cycles.

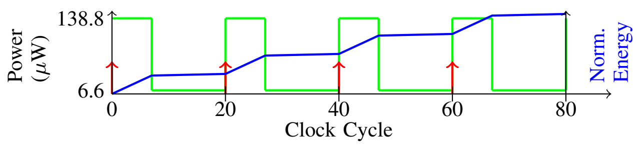


Figure 9: Energy and average power consumption for a self-powered `FIR` design stimulated with $U = 20\,\%$ utilization and $I = 100\,\%$ intermittency. The red arrows indicate the arrival of input activations every 20 clock cycles. The green line illustrates the average power consumption during the sleep and active stages, while the blue line represents the total energy consumed.

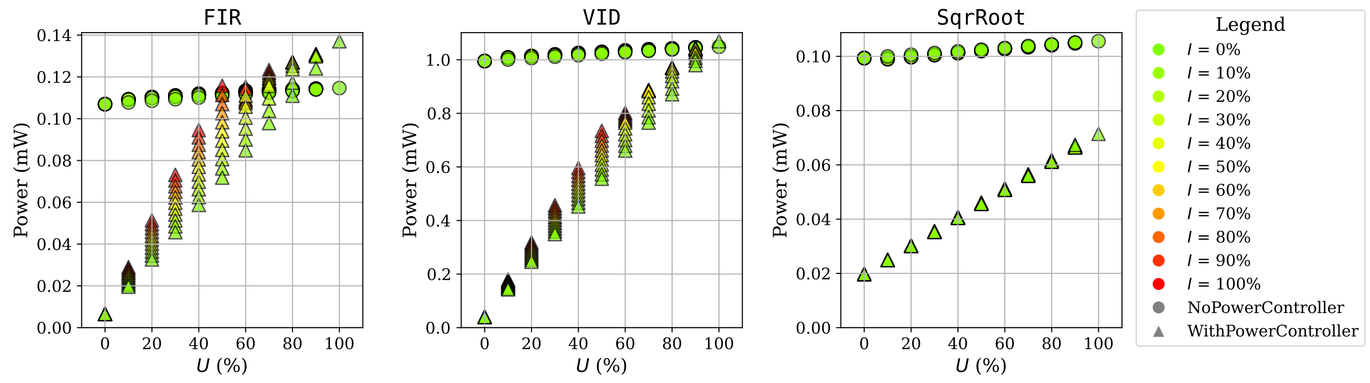


Figure 10: Power consumption of the test applications `FIR`, `VID`, and `SqrRoot` using stimuli for different values of U and I. The triangles represent the avg. power of the self-powered actor designs, while the circles represent the avg. power of the corresponding reference design without any power control.

C. *Power and Energy Reduction Analysis using Clock Gating*

For ASIC implementations of self-powering dataflow networks with tight dynamic power requirements and low static power (leakage), clock gating is an appealing technique. To evaluate its benefits, the resulting power averaged over the period $T(U)$ is depicted in Fig. 10 without clock gating (circles) and with clock gating (triangles) for each application and given values of $U, I \in]0, 100]\,\%$ in steps of $10\,\%$.

For $U = 100\,\%$, clock gating induces a power overhead of $20\,\%$ and $2\,\%$ for the `FIR`, respectively, `VID` application due to the additional power management circuitry, while a power reduction of $32\,\%$ is observed for the `SqrRoot` application because some actors can sleep during one activation even.

For the case of zero activations (idle case) corresponding to the border case of $U = 0\,\%$, power savings of $94\,\%$, $96\,\%$, and $80\,\%$ are obtained for the applications `FIR`, `VID`, and `SqrRoot`, respectively. A clear trend is observed, showing increased power savings with reduced utilization as the actors can sleep for longer durations. For the applications `FIR` and `VID`, decreasing intermittency also leads to increasing power savings as actors sleep less frequently but for correspondingly longer intervals, reducing the overhead due to sleep-wakeup cycles. For the application `SqrRoot`, no such influence can be observed as the actors a_2 to a_4 always go to sleep in each approximation step (cf. Fig. 2) as there is only a single token in the actor cycle a_2, a_3, a_4.

In summary, we can say that for actor networks with a DAG topology (i.e., no directed cycles), there are no throughput degradations observable through the introduction of self powering, only light increases in latency per activation.

VI. Conclusion and Outlook

In this paper, we presented concepts and an automatic approach to the hardware synthesis of self-powering dataflow networks. The resulting actors are automatically put into sleep mode if not fireable and are only powered on when fireable. For different test applications, self-powering can be supported by either clock gating to save dynamic power and/or power gating to also reduce static power.

Due to space limitations, we were only able to present power and thus energy savings for clock gating. Results on power and energy savings for power-gated actor implementations will be presented as part of future work.

References

[1] E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow," *Proc. IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[2] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, "Cyclo-Static Data Flow," *IEEE Trans. Signal Process.*, vol. 44, no. 2, pp. 397–408, 1996.

[3] J. Falk, C. Haubelt, C. Zebelein, and J. Teich, "Integrated Modeling Using Finite State Machines and Dataflow Graphs," in *Handbook of Signal Processing Systems*, S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, Eds. Springer, 2013, pp. 975–1013.

[4] S. A. Edwards, "The Challenges of Hardware Synthesis from C-Like Languages," in *2005 Design, Automation and Test in Europe Conference and Exposition (DATE 2005), 7-11 March 2005, Munich, Germany*. IEEE Computer Society, 2005, pp. 66–67.

[5] J. Keinert, M. Streubühr, T. Schlichter, J. Falk, J. Gladigau, C. Haubelt, J. Teich, and M. Meredith, "SystemCoDesigner — An Automatic ESL Synthesis Approach by Design Space Exploration and Behavioral Synthesis for Streaming Applications," *ACM Trans. Design Autom. Electr. Syst.*, vol. 14, no. 1, pp. 1:1–1:23, 2009.

[6] J. W. Janneck, I. D. Miller, D. B. Parlour, G. Roquier, M. Wipliez, and M. Raulet, "Synthesizing Hardware from Dataflow Programs - An MPEG-4 Simple Profile Decoder Case Study," *J. Signal Process. Syst.*, vol. 63, no. 2, pp. 241–249, 2011.

[7] J. Eker and J. W. Janneck, "Dataflow Programming in CAL - Balancing Expressiveness, Analyzability, and Implementability," in *Proc. of the 46th Asilomar Conference (ACSCC)*, M. B. Matthews, Ed. IEEE, 2012, pp. 1120–1124.

[8] J. Falk, C. Haubelt, and J. Teich, "Efficient Representation and Simulation of Model-Based Designs in SystemC," in *Proc. of the Forum on Specification and Design Languages (FDL)*, vol. 6, 2006.

[9] L. Thiele, K. Strehl, D. Ziegenbein, R. Ernst, and J. Teich, "FunState–An Internal Design Representation for Codesign," in *Proc. of ICCAD*. IEEE, 1999, pp. 558–565.

[10] M. Baird, Ed., *IEEE Standard 1666-2005 SystemC Language Reference Manual*. New Jersey, USA: IEEE Standards Association, 2005.

[11] C. Lattner, "Keynote Talk: LLVM and Clang: Advancing Compiler Technology," in *Proc. Free and Open Source Developers European Meeting*, ser. FOSDEM'11, Feb. 2011.

[12] J. Falk, J. Keinert, C. Haubelt, J. Teich, and S. Bhattacharyya, "A Generalized Static Data Flow Clustering Algorithm for MPSoC Scheduling of Multimedia Applications," in *Proc. of EMSOFT*, Oct. 2008, pp. 189–198.

[13] S. EDA, "Questa Advanced Simulator," https://eda.sw.siemens.com/en-US/ic/questa/, 2024.

[14] I. Synopsys, "Synopsys PrimePower," https://www.synopsys.com/, 2024.