Design and Implementation of Reconfigurable Asynchronous Pipelines

Alessandro de Gennaro[†,§], Danil Sokolov[†], Andrey Mokhov[†]
alessandro.de-gennaro@tiempo-secure.com, {danil.sokolov, andrey.mokhov}@newcastle.ac.uk
[†]*School of Engineering, Newcastle University, UK*
[§]*Tiempo Secure - SAS, Montbonnot-Saint-Martin, France*

***Abstract*—Pipelining is a widely used approach to the design of high-throughput computation systems, where the slowest component is decomposed into a set of sequentially connected parts that are executed in parallel on successive items of the incoming dataflow. Such *dataflow pipelines* are often designed to be *dynamically reconfigurable* to process data items differently depending on their contents and/or to adjust to the application requirements in runtime.**

Reconfigurable synchronous pipelines are widely used and well studied, and are supported by industrial EDA tools. On the other hand, *reconfigurable asynchronous pipelines* received much less attention and their industrial adoption is low due to the lack of mature automation support. In this paper we present a model and a tool support for the design and verification of reconfigurable asynchronous pipelines. The tool is open-source and is available as a plugin for the WORKCRAFT toolset. We validate the presented approach by designing and fabricating a test chip (TSMC 90nm), that demonstrates the benefits and costs of dynamic reconfigurability, as well as highlights the resilience of asynchronous pipelines.

I. INTRODUCTION

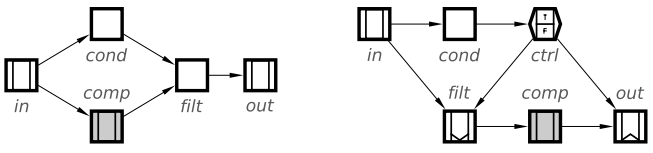
Dataflow pipelines are a simple and powerful tool for the design of high-throughput computation systems. Given a system with an identified performance bottleneck (its slowest component), one can decompose the latter into a sequential composition of smaller parts, and run them all in parallel on different items of the incoming flow of data. The result is a higher throughput at the cost of increased latency, which is an acceptable trade-off for many applications [1].

Real pipelines are often non-linear. They need to be dynamically reconfigurable to process diverse data items differently, or adjust to the current environmental conditions. One example of reconfigurable pipelines is machine learning networks, e.g. described using Google's TensorFlow [2]. Dynamic reconfigurability is essential for the performance and efficient use of resources in machine learning pipelines, as emphasised by Martín Abadi in his recent ICFP 2016 keynote [3]. TensorFlow is supported by Google's hardware *tensor processing units*, that can be combined into distributed computation systems. *Spatial computing* [4] is another example of distributed dataflow graphs employed in application-specific high-performance data analysis. Such large-scale dataflow graphs must necessarily be asynchronous at the top level. On the other side of the spectrum, there are IoT nodes and mixed-signal microcontrollers that achieve higher energy-efficiency by asynchronous event-driven processing [5]. These application areas motivate our research in *reconfigurable asynchronous pipelines*.

Synchronous dataflow pipelines have been studied since 1980s [1], and are well supported by formal models and mainstream EDA tools. An example of a formalism for the specification, optimisation and verification of reconfigurable synchronous pipelines is xMAS [6].

Asynchronous non-reconfigurable pipelines have also been extensively studied, e.g. [7] provides an in-depth overview of existing hardware implementation styles, while [8] introduces *Static Dataflow Structures* (SDFS), a formal behavioural model for non-reconfigurable asynchronous pipelines further developed in [9]. As we show in Section II, SDFS cannot adequately model dynamic pipeline reconfiguration.

The main result of this paper is an extension of SDFS to a more general formalism of *Dataflow Structures* (DFS), which is capable of capturing both static and dynamically reconfigurable asynchronous pipelines. As the motivating example for the DFS formalism, consider an asynchronous pipeline that applies a computationally expensive pipelined function comp only to those data items that satisfy an easily-checked condition cond, e.g. computing a square root only for non-negative numbers. Figure 1a shows an SDFS model [8] of such a pipeline, which only supports RTL-style *logic* and *register* components (we show the comp pipeline as a shaded register for simplicity). Crucially, both cond and comp must be executed before filtering unneeded results with filt and producing the output. This limits the performance and degrades the power consumption of the pipeline to the worst-case scenario, which is clearly undesirable.



(a) SDFS model. (b) DFS model.

Fig. 1: Conditional application of function comp.

To adequately model such dynamic behaviour we introduce the DFS formalism that uses three new types of registers, namely *control*, *push*, and *pop*, for orchestrating the flow of data items in reconfigurable pipelines, see Figure 2.

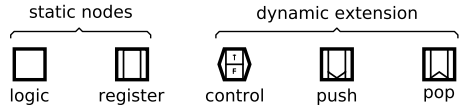


Fig. 2: Five types of DFS nodes.

Figure 1b shows a DFS model of the above example, which applies the cond predicate to the incoming data item, or simply *token*, and produces the True or False token in ctrl that guards the time-consuming comp function. In case of the False token, the input token is consumed by the push register filt and an empty token is produced by the pop register out, thus bypassing comp. Otherwise, if ctrl contains True, the push and pop act as static registers, and the token propagates to the output via the comp pipeline. We can specify this by the expression in $\rightarrow [cond]$comp $\rightarrow$ out using the algebra of *Parameterised Graphs* [20], which can be used as a high-level specification language for reconfigurable pipelines.

The semantics of DFS components is expressed using *Petri Nets* [10] thus enabling the reuse of established theory and tools developed by the Petri Net community. In particular, our DFS software support is implemented as a plugin for the WORKCRAFT framework [11] that uses Petri Nets as a common behaviour description language for integrating a set of backend tools, such as PETRIFY [12] and MPSAT [13].

We also present a DFS-based design methodology and a set of generic pipeline stage components for building reconfigurable asynchronous pipelines (released under MIT license).

For circuit implementation we rely on mapping of model elements into predefined hardware components, similar to those for handshake circuits in the syntax-directed translation [14] that is employed in Tangram [15] and Balsa [16]. Presently this is the only way to automate construction of large asynchronous circuits: formal synthesis is too computationally expensive and templated synthesis [17] remains an art rather than an automatable procedure.

The methodology was validated by designing, fabricating and testing a reconfigurable ASIC accelerator for *ordinal pattern encoding* (OPE) [18]. The chip provides real-life data for analysing benefits and costs of dynamic reconfigurability, and the resilience of asynchronous pipelines in unpredictable environmental conditions, such as unstable voltage supply.

The main contributions of this paper[1] are as follows:

- **A formal definition of the DFS model** and its Petri Net behavioural semantics (Section II) that enables the reuse of existing verification and synthesis methods.
- **An open-source EDA tool** available as a plugin for the WORKCRAFT toolset (https://www.workcraft.org/), which provides a complete DFS-based design flow for reconfigurable asynchronous pipelines (Section II-D).
- **A DFS-based design methodology** for reconfigurable asynchronous pipelines, and its evaluation by the design and verification of the OPE accelerator (Section III).
- **An ASIC prototype** implementing both static and dynamically reconfigurable OPE accelerators as a validation of the presented approach (Section IV).
- **Evaluation of DFS formalism** in the relation to existing dataflow-based models of computation (Section V).

[1]This paper is an extension of [19] and includes the following changes. An extendend description of the DFS behavioural semantics where all DFS components are formally defined, in Sections II-A and II-B. New details on the hardware implementation of DFS pipelines, in Section III-B. Details on the experiment objectives and additional test results, in Sections IV-A and IV-C. An extended related work Section V on Boolean-controlled dataflow models. Our considerations on the applicability of the presented design flow to existing asynchronous communication protocols and timing models, in Section VI.

$$C(l) = C_{\uparrow}(l) \vee \overline{C_{\downarrow}(l)} \wedge C(l)$$
$$C_{\uparrow}(l) = \bigwedge_{k \in \bullet l \cap L} C(k) \wedge \bigwedge_{q \in \bullet l \cap R} M(q)$$
$$C_{\downarrow}(l) = \bigwedge_{k \in \bullet l \cap L} \overline{C(k)} \wedge \bigwedge_{q \in \bullet l \cap R} \overline{M(q)}$$

preset logic preset registers

(a) Logic nodes.

$$M(r) = M_{\uparrow}(r) \vee \overline{M_{\downarrow}(r)} \wedge M(r)$$
$$M_{\uparrow}(r) = \bigwedge_{k \in \bullet r \cap L} C(k) \wedge \bigwedge_{q \in \star r} M(q) \wedge \bigwedge_{q \in r \star} \overline{M(q)}$$
$$M_{\downarrow}(r) = \bigwedge_{k \in \bullet r \cap L} \overline{C(k)} \wedge \bigwedge_{q \in \star r} \overline{M(q)} \wedge \bigwedge_{q \in r \star} M(q)$$

preset logic R-preset R-postset

(b) Static registers.

Fig. 4: SDFS behavioural semantics.

II. DATAFLOW STRUCTURES MODEL

Formally, a dataflow structure DFS is defined as a triple $\mathsf{DFS} = \langle V, E, M_0 \rangle$, where:

- $V = L \cup R$ is the set of *logic* and *register* nodes;
- $E \subseteq V \times V$ is the set of *arcs* (interconnect);
- $M_0 = R \rightarrow \{0, 1\}$ is the *initial marking* of registers.

The *preset* and *postset* of a node $x \in V$ are defined as:

- $\bullet x = \{y : (y, x) \in E\}$ is the *preset* of a node x;
- $x\bullet = \{y : (x, y) \in E\}$ is the *postset* of a node x.

A *logic path* $\delta(s, t)$ between nodes s and t is a non-empty sequence of edges $(z_{i-1}, z_i) \in E, i \in [1..n]$, such that $z_0 = s$, $z_n = t$, and $z_i \in L$ for $0 < i < n$.

The *R-preset* and *R-postset* of a node $x \in V$ are:

- $\star x = \{y \in R : \exists \delta(y, x)\}$ is the *R-preset* of a node x;
- $x \star = \{y \in R : \exists \delta(x, y)\}$ is the *R-postset* of a node x.

A. Static nodes

Figure 4 shows the equations that orchestrate the SDFS behavioural semantics. A logic node $l \in L$ models a combinational dataflow component [8]. It can be *evaluated* when the logic nodes in its preset are evaluated and the registers in its preset are *marked* (i.e. contain valid data). Symmetrically, a logic node can be *reset* when the logic nodes in its preset are reset and the registers in its preset are *unmarked* (i.e.

$$C(l) = C_{\uparrow}(l) \vee \overline{C_{\downarrow}(l)} \wedge C(l)$$

$$C_{\uparrow}(l) = \bigwedge_{k \in \bullet l \cap L} C(k) \wedge \bigwedge_{r \in \bullet l \cap (R \setminus R_{push})} M(r) \wedge \bigwedge_{p \in \bullet l \cap R_{push}} M_t(p)$$

$$C_{\downarrow}(l) = \bigwedge_{k \in \bullet l \cap L} \overline{C(k)} \wedge \bigwedge_{r \in \bullet l \cap (R \setminus R_{push})} \overline{M(r)} \wedge \bigwedge_{p \in \bullet l \cap R_{push}} \overline{M_t(p)}$$

preset logic | preset registers but push | preset push registers

(a) Logic nodes.

$$M(r) = M_{\uparrow}(r) \vee \overline{M_{\downarrow}(r)} \wedge M(r)$$

$$M_{\uparrow}(r) = \bigwedge_{l \in \bullet r \cap L} C(l) \wedge \bigwedge_{q \in \star r \cap (R \setminus R_{push})} M(q) \wedge \bigwedge_{p \in \star r \cap R_{push}} M_t(p) \wedge \bigwedge_{q \in r \star \cap (R \setminus R_{pop})} \overline{M(q)} \wedge \bigwedge_{p \in r \star \cap R_{pop}} \overline{M_t(p)}$$

$$M_{\downarrow}(r) = \bigwedge_{l \in \bullet r \cap L} \overline{C(l)} \wedge \bigwedge_{q \in \star r \cap (R \setminus R_{push})} \overline{M(q)} \wedge \bigwedge_{p \in \star r \cap R_{push}} \overline{M_t(p)} \wedge \bigwedge_{q \in r \star \cap (R \setminus R_{pop})} M(q) \wedge \bigwedge_{p \in r \star \cap R_{pop}} M_t(p)$$

preset logic | R-preset registers but push | R-preset push registers | R-postset registers but pop | R-postset pop registers

(b) Static registers.

$$M(d) = M_t(d) \vee M_f(d)$$

$$M_t(d) = M_{\uparrow}(d) \ \wedge \ M_t^s(d) \ \vee \ \overline{M_{\downarrow}(d)} \ \wedge \ M_t(d)$$

$$M_f(d) = M_{\uparrow}(d) \ \wedge \ M_f^s(d) \ \vee \ \overline{M_{\downarrow}(d)} \ \wedge \ M_f(d)$$

(c) Dynamic registers.

preset control registers

$$M_t^s(d) = \bigwedge_{c \in \bullet^1 d} M_t(c) \wedge \bigwedge_{c \in \bullet^0 d} M_f(c)$$

$$M_f^s(d) = \bigwedge_{c \in \bullet^1 d} M_f(c) \wedge \bigwedge_{c \in \bullet^0 d} M_t(c)$$

regular arcs | inverting arcs

(d) Conditions for transparent and opaque push/pop.

Fig. 5: DFS behavioural semantics.

contain no data). The *evaluation state* $C(l)$ of a logic node $l \in L$ can be defined using the evaluate $C_{\uparrow}(l)$ and reset $C_{\downarrow}(l)$ functions (similar to the set/reset functions in the SR latch equation $Q = S \vee \overline{R} \wedge Q$), see Fig. 4a.

A static register node $r \in R_{stc}$ models a sequential dataflow component [8]. It can accept a token of data when its preset logic nodes are evaluated, R-preset registers are marked, and R-postset registers are unmarked. Symmetrically, a token can leave a register when its preset logic is reset, the R-preset is unmarked, and the R-postset is marked. The *marking state* $M(r)$ of register $r \in R$ is defined in Fig. 4b.

B. Dynamic extension

In addition to the static registers R_{stc}, the DFS model introduces *control*, *push*, and *pop* types of register nodes: $R_{stc} \cup R_{ctrl} \cup R_{push} \cup R_{pop} = R$. The equations that orchestrate their behavioural semantics are in Figure 5.

Control registers can only contain True or False tokens. They can propagate the regular or inverting polarity of their hold tokens by using *regular* (—►) or *inverting* (—►○) arcs. Control registers in the inverting arc preset of a node x (i.e. connected by inverting arcs) are denoted as $\bullet^0 x$, while those in regular arc preset, as $\bullet^1 x$.

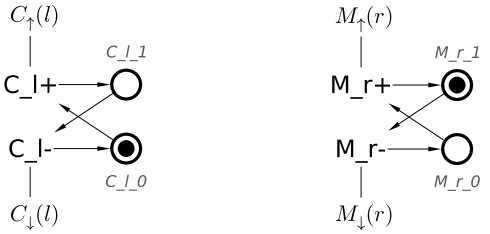
A push/pop is said to be *transparent* if all control registers in its regular arc preset hold the True tokens, and in inverted arc preset hold the False tokens. A transparent push/pop behaves as a static register.

A push/pop is *opaque* if it has False tokens in its regular arc preset and True tokens in its inverting arc preset. An opaque push consumes and destroys an incoming token (behaves as a token sink), while opaque pop produces an empty token for its postset nodes (behaves as a token source).

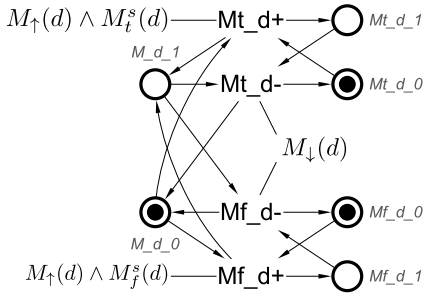
In case of a mismatch, i.e. when a push/pop sees both True and False tokens in its preset, the register is disabled, which may lead to a deadlock. The reachability of such problematic states needs to be formally verified, which has been automated in our design flow.

With the introduction of dynamic registers, the behavioural semantic of logic nodes (Fig. 4a) and static registers (Fig. 4b) are refined as shown in Fig. 5a and Fig. 5b, respectively. Intuitively, logic nodes are passive to the handshake mechanism: they are evaluated when all their inputs are available, and reset when their inputs do not contain valid data. Similarly, a register can store a data value when all its inputs are available and its R-postset registers contain no tokens (an opaque pop can hold a token though); and can be reset when the stored data has been propagated to its R-postset. This semantics models the *4-phase handshake protocol* [29].

For simplicity we say that push/pop handle True tokens when transparent, and False tokens when opaque. The function $M_t(p)$ determines if a push or pop register $p \in R_{push} \cup R_{pop}$ is marked with a True token (i.e. it operates as a static register). Symmetrically, the function $M_f(p)$



(a) Logic nodes. (b) Static registers.



(c) Dynamic registers.

Fig. 6: Petri Net semantics of DFS nodes.

determines if p received a token is marked with a False token. We use these functions to represent the behaviour of dynamic registers $d \in R_{ctrl} \cup R_{push} \cup R_{pop}$, see Fig. 5c.

The dynamic register marking $M(d)$ is defined as a non-deterministic choice between being marked with a True token $M_t(d)$ or a False token $M_f(d)$. The functions $M_t^s(d)$ and $M_f^s(d)$ (defined in Fig. 5d) determine if a dynamic register is transparent or opaque, respectively, by checking its preset control registers $c \in \bullet d \cap R_{ctrl}$.

The DFS model can also implement a Boolean algebra on True and False tokens using control nodes with AND/OR/C-element logic of token synchronisation, which allows modelling complex reconfiguration strategies. This feature is outside the scope of the paper.

C. Petri Net semantics

We express the execution semantics of DFS components using *Petri Nets* (PN) [10] with the read-arcs extension [21], similar to so-called *Circuit Petri Nets* [22]. The PN semantics enables the reuse of established theory and tools for the formal verification and synthesis available within WORKCRAFT. Other general-purpose formalisms can be used to capture DFS semantics, for example finite state machines, process algebra [23], Event-B [24] or dependently-typed languages, such as Agda [25] or Idris [26]. However, the WORKCRAFT framework that we use for the design of asynchronous dataflow pipelines does not currently provide any verification backends for these formalisms and languages, hence our pragmatic choice of Petri Nets.

A static node can be characterised by a single Boolean variable representing its state, e.g. C_l characterises the evaluation state of a logic node $l \in L$, and M_r – the marking state of a register $r \in R_{stc}$. A variable x is translated into a pair of PN places x_0 and x_1 representing its 0 and 1 states, respectively. One of these places is marked with a token to reflect the initial state of x. Transitions x+ and x- that represent the variable changes are inserted consistently between the places, thus forming x_0 $\rightarrow$ x+ $\rightarrow$ x_1 and x_1 $\rightarrow$ x- $\rightarrow$ x_0 paths. Enabledness of these transitions is restricted by means of read-arcs from the other variables' places, according to the set/reset functions of the node state equations in Fig. 5a and Fig. 5b. Figure 6a shows a PN snippet for a logic node in the reset state and Figure 6b – for a marked register.

For dynamic registers $d \in R_{ctrl} \cup R_{push} \cup R_{pop}$ two additional variables Mt_d and Mf_d are used, representing its True and False marking, as shown in Figure 6c. Note that transition M_d+ is refined into a pair of mutually-exclusive transitions Mt_d+ and Mf_d+. Similarly, M_d- is refined into Mt_d- and Mf_d-.

The DFS model of our first motivating example (Figure 1b), is translated into a PN shown in Figure 7. Notice that transitions Mt_ctrl+ and Mf_ctrl+ can be enabled simultaneously, thus representing a non-deterministic choice for the evaluation result of the cond logic. The choice between Mt_filt+ and Mf_filt+ on the other hand is determined by Mt_ctrl_1 and Mf_ctrl_1 places that can never be marked together.

D. Design automation

The design of DFS models has been automated within the open-source WORKCRAFT toolset [11]. It provides a cross-platform GUI for convenient editing, interactive simulation and performance analysis of DFS pipelines. For computationally intensive formal verification the DFS models are mechanically translated into Petri Nets and passed to MPSAT backend [13]. MPSAT enables verifying DFS models for the standard properties (such as deadlock) and custom functional properties (such as hazards) expressed in Reach language [27].

A screenshot in Figure 8 demonstrates the performance analysis of a reconfigurable pipeline in the WORKCRAFT graphical environment. The tool reports the throughput of the slowest cycles and highlights the bottleneck nodes in each cycle. The user can analyse the difference in cycles' throughput and balance them by adjusting the number of tokens, adding registers to buffer the flow of tokens, and applying advanced performance optimisation techniques, such as wagging [28].

A verified and optimised DFS model can be automatically translated into an asynchronous circuit netlist by directly mapping its nodes into pre-built components and connecting them according to the dataflow arcs. A library of such components determines the circuit implementation style – see [7] for a comprehensive overview of popular implementation styles of asynchronous pipelines. Alternatively, it is also possible to synthesise a fully custom implementation by extracting the underlying PN representation of a DFS model and using standard PN circuit synthesis tools supported by WORKCRAFT. The circuit is subsequently exported as a Verilog netlist to be used

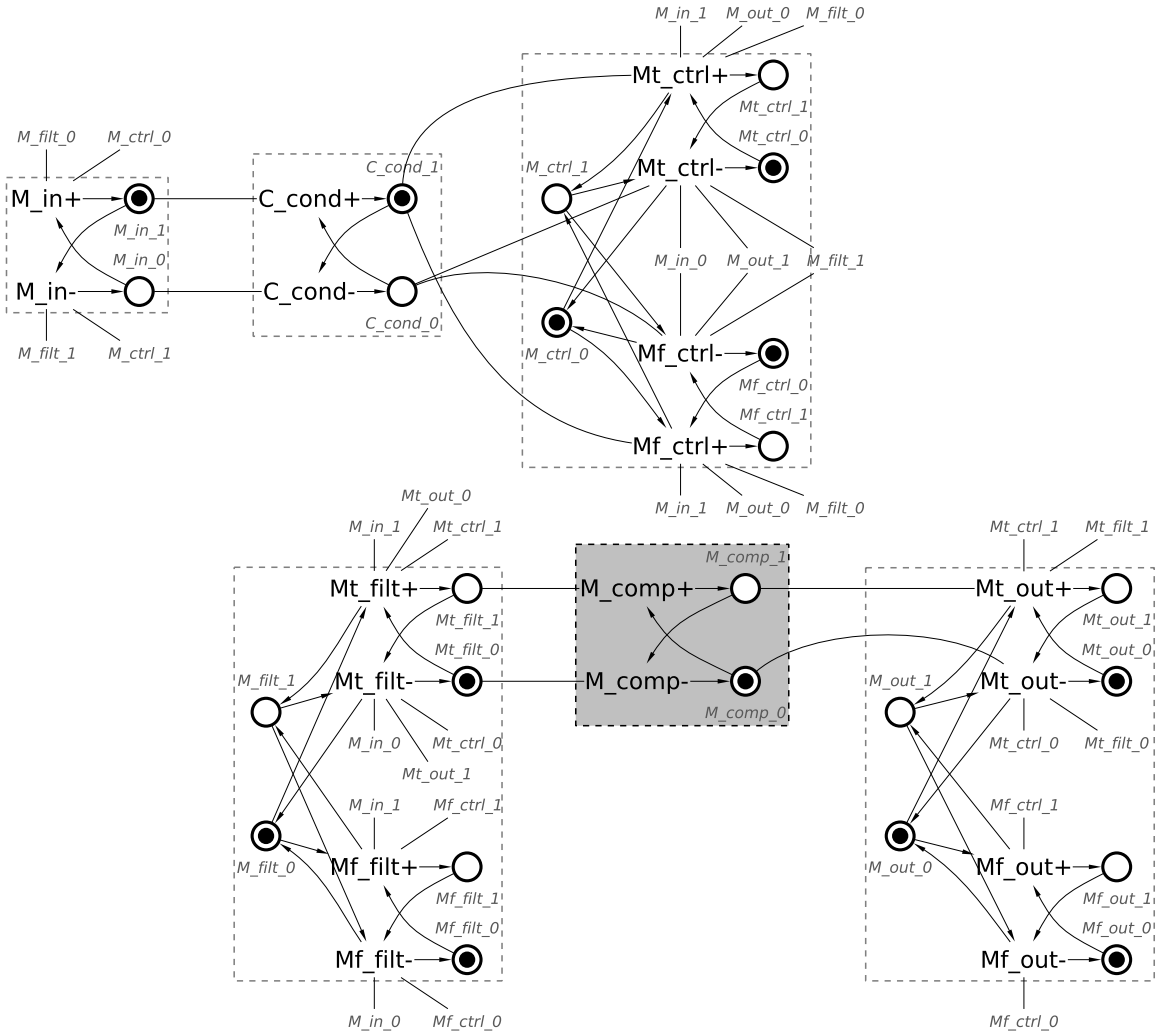


Fig. 7: Petri Net semantics of the DFS in Figure 1b.

in a conventional backend flow. Support of other hardware description languages, such as VHDL and Balsa [16], is subject of future work.

III. Reconfigurable pipelines

In this section we present a DFS-based methodology for modelling reconfigurable asynchronous pipelines, and also validate it on a case study.

Figure 9a shows a generic pipeline structure comprising N stages, who interface to each other via *local channels* (dashed arcs), and are connected to the common input in and aggregated output out via *global channels* (solid arcs).

A DFS model for a pipeline stage is shown in Figure 9b. It applies a function f to the tokens in the local_in1 register (input data from the previous stage local output) and produces a token in the local_out register (local output data for the next stage). The produced token, paired with the common input token in global_in and the global output of the previous register in local_in2, are passed to a function g, which produces a global_out token, used to aggregate the output of all stages.

One typical reconfiguration scenario for such a pipeline is to change its *depth* (i.e. the number of stages) depending on the application requirements. We design a reconfigurable generic pipeline that is capable of using a given number of initial stages, bypassing the remaining ones. Figure 9c shows our DFS design of a reconfigurable pipeline stage. The stage local input is implemented as a pair of push registers local_in1 and local_in2 controlled by the local_ctrl structure. The global_in and global_out are push and pop registers, respectively, controlled by the global_ctrl structure. Both local_ctrl and global_ctrl are 3-register loops (the minimum number of registers required for a token oscillation). To include a stage into the reconfigurable pipeline, these control loops need to be initialised with the True tokens; to exclude it – with the False tokens. Note that a token starts oscillating in local_ctrl only if the previous stage is included in the pipeline and global_in operates as a static register – this is done to prevent the stage operation when the previous stage is inactive.

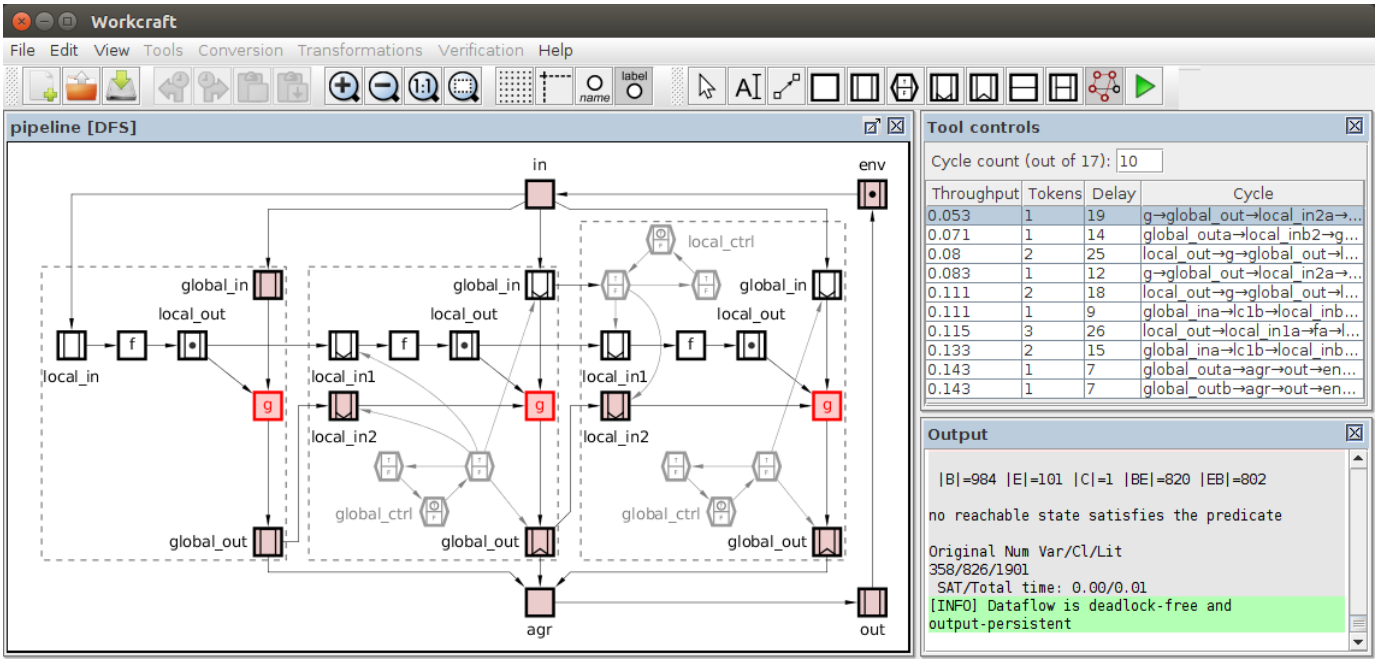
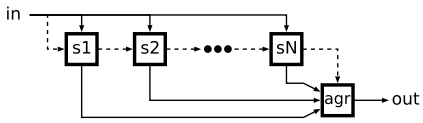
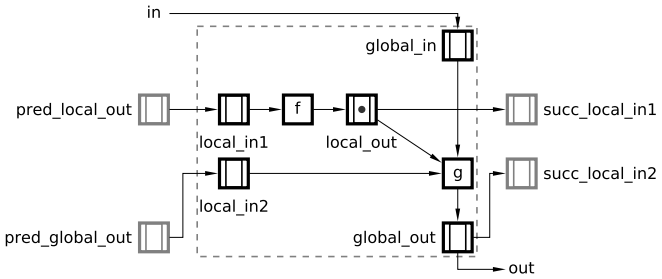


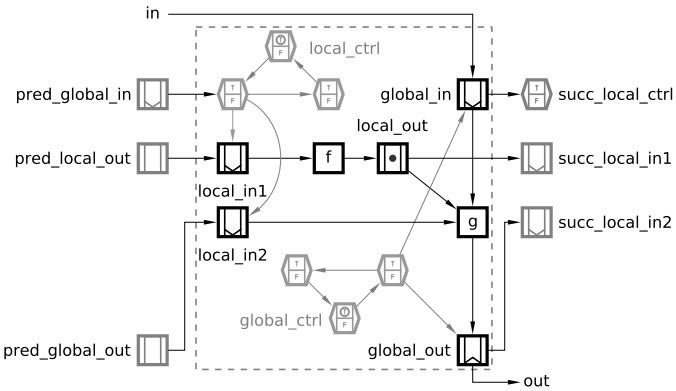
Fig. 8: Performance analysis of DFS in WORKCRAFT.



(a) Generic N-stage pipeline.



(b) Static pipeline stage.



(c) Reconfigurable pipeline stage.

Fig. 9: Pipeline with local and global stage interfaces.

A. Case study

We apply the proposed methodology to the design of an asynchronous dataflow accelerator for reconfigurable ordinal pattern encoding (OPE) [18]. It ranks the last N items in an incoming data stream[2], a common task in statistical analysis with a wide range of applications: from stock market prediction to medical data analysis. The OPE case study is an interesting benchmark for our DFS modelling methodology because it requires reconfigurability and is conventionally implemented as a dataflow pipeline.

The OPE functionality is best explained by an example. Consider the stream of numbers $(3, 1, 4, 1, 5, 9, 2, 6)$ and the window size $N = 5$. The table below shows the windows starting at different indices within the stream and the corresponding rank lists:

Index	*Window*	*Rank list*
1	$(3, 1, 4, 1, 5)$	$(3, 1, 4, 2, 5)$
2	$(1, 4, 1, 5, 9)$	$(1, 3, 2, 4, 5)$
3	$(4, 1, 5, 9, 2)$	$(3, 1, 4, 5, 2)$
4	$(1, 5, 9, 2, 6)$	$(1, 3, 5, 2, 4)$

If the window size is increased to $N = 6$, the following results are obtained instead:

Index	*Window*	*Rank list*
1	$(3, 1, 4, 1, 5, 9)$	$(3, 1, 4, 2, 5, 6)$
2	$(1, 4, 1, 5, 9, 2)$	$(1, 4, 2, 5, 6, 3)$
3	$(4, 1, 5, 9, 2, 6)$	$(3, 1, 4, 6, 2, 5)$

[2]The *rank* of an item in a list is the position the item ends up at after sorting the list. For example, ranks of items in the list $(2, 0, 1, 7)$ are $(3, 1, 2, 4)$.

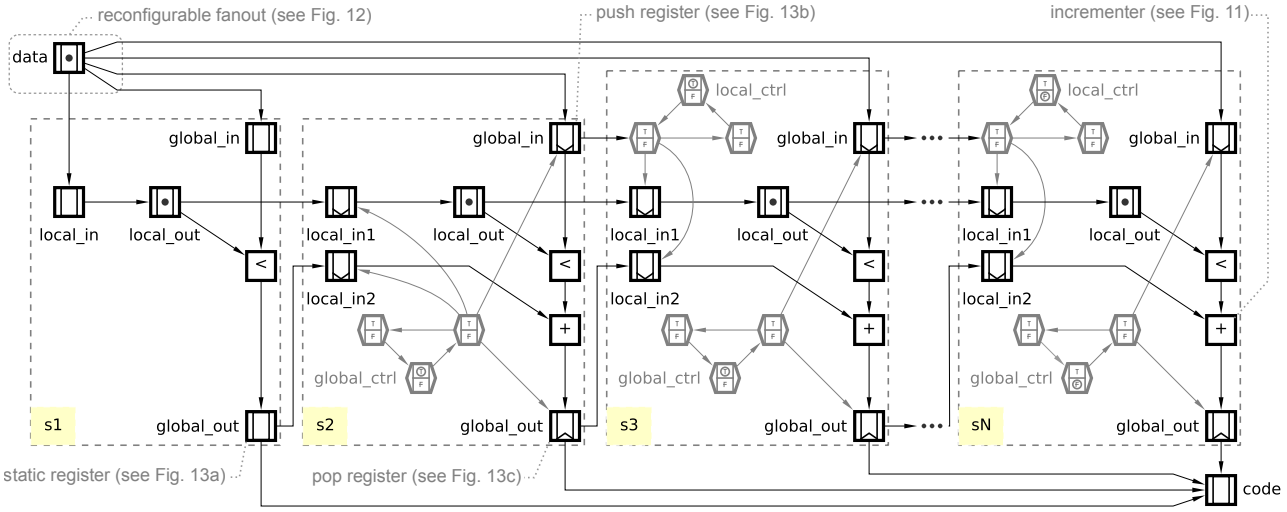
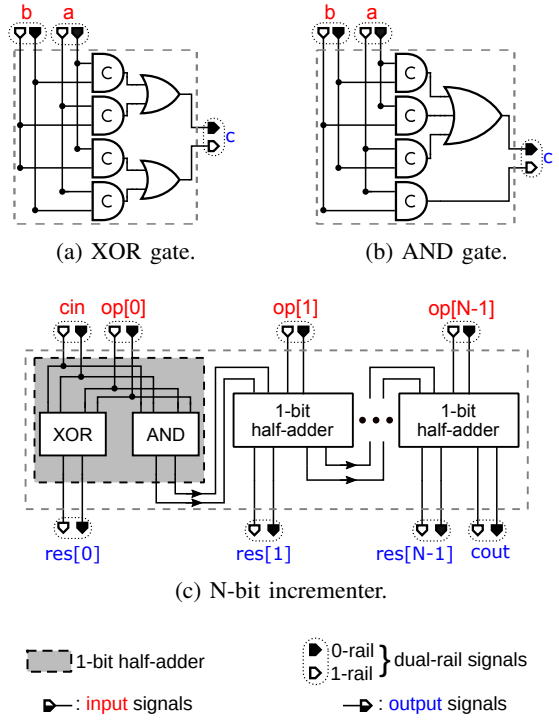


Fig. 10: DFS model of the reconfigurable OPE pipeline (from 1 to N stages) that corresponds to the OPE window size.



(a) XOR gate. (b) AND gate.

(c) N-bit incrementer.

Fig. 11: Implementation of an asynchronous dual-rail N-bit incrementer, with NCL-D style gates.

The OPE pipeline is designed to compute such rank lists very efficiently: ranks of elements in a window are calculated concurrently and the produced rank list is reused when processing the next window. Users of OPE engines often try multiple window sizes N (via reconfiguration) to discover hidden patterns in a stream of data. Our implementation is based on the synchronous pipeline design presented in [18].

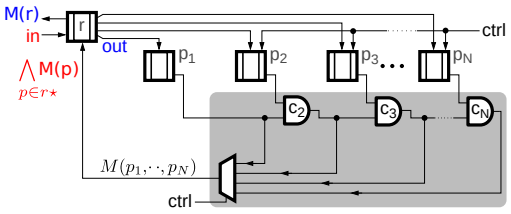
Figure 10 shows our DFS model of a reconfigurable OPE pipeline. The first stage s1 is always included and is therefore implemented in the static style; the remaining stages are reconfigurable. Note that the stage s2 is optimised by reusing global_ctrl to control both the local and global interfaces, which is possible because s1 is always included in the pipeline and its global_in is a register, not a push.

Using the developed WORKCRAFT plugin, the DFS model of the reconfigurable OPE pipeline can be visually simulated and formally verified at the abstract technology-independent level where data is represented by abstract tokens. Several cases of deadlock and non-persistent behaviour (mostly due to incorrect initialisation of control registers) were identified, analysed and corrected during the design process.

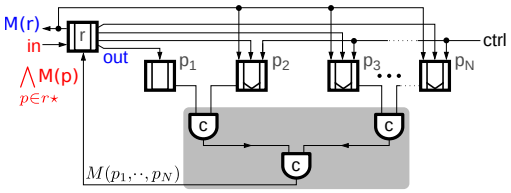
B. Hardware implementation

The DFS model was translated into a circuit implementation netlist using a library of pre-built NCL-D style asynchronous dual-rail components (comparator, adder, and a set of registers) that rely on 4-phase communication protocol [29]. A conventional flow, in compliance with the quasi-delay-insensitive (QDI) timing model [8], was subsequently employed for technology mapping, layout, and place-and-route tasks. In this section, we show the implementation of a subset of components implemented in the OPE pipeline (shown in Figure 10).

As an example of a combinatorial logic node, the circuit behind the incrementer embedded in the OPE stages (node '+' in Figure 10) is shown in Figure 11. Its purpose is to increment the result of the previous stage (op operand contained in the local_in2 register) by the comparator result (cin operand in the node '<'), which can either be a logic 0 or 1. The incrementer is implemented as a chain of half-adders (see Figure 11c) made of NCL-D XOR and AND gates (see Figures 11a-b). In the schematics shown in this section, dual-rail signals have their pins coloured as black and white pairs, the former represents the 0-rail and the latter represents the 1-rail, see legend at the bottom of Figure 11. The legend also shows the symbols of input and output pins.



(a) Reconfigurable interconnections.



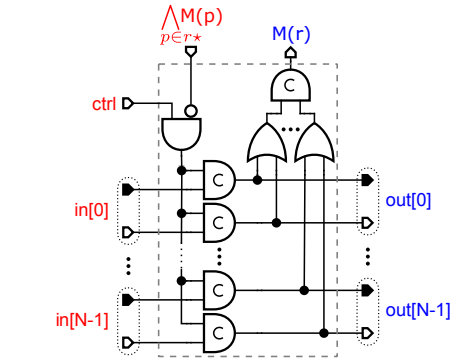
(b) Distributed reconfiguration.

Fig. 12: Asynchronous reconfigurable fanout implementations.

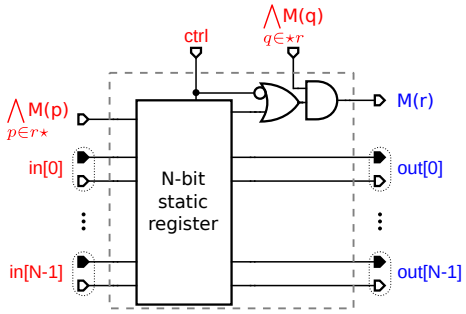
While logic nodes are passive to the handshake mechanism, register nodes have an active role. Their implementation is at the core of asynchronous reconfiguration mechanisms. Figure 12 shows two approaches to implement the handshake between r (data node in Figure 10) and its N reconfigurable fanout registers p (global_in nodes) representing the pipeline stages. The pipeline can be shortened by disabling the latest push registers, e.g. if the pipeline depth is 2: $\{p_1, p_2\}$ are enabled, if it is 3: $\{p_1, p_2, p_3\}$ are enabled, and so on.

In the approach in Figure 12a, all registers are mapped to the *N-bit static register* implementation that is shown in Figure 13a and described as follows. Static registers are enabled (the operation mode ON) when the control input ctrl is set to 1, otherwise they are disabled (mode OFF). When ON, these registers synchronise with the rest of the circuit by relying on the R-postset marking state M(p) and the produced register marking state M(r), according to the 4-phase dual-rail asynchronous communication protocol as described in [8]. When OFF, input tokens are masked and registers are practically disabled. Notice that in both Figures 12a and 12b, the register p_1 is always ON, and its corresponding ctrl input signal is omitted. In the approach in Fig. 12a, the reconfiguration is managed at the interconnection between registers, i.e. the fanout registers p are connected to r through a daisy chain of C-elements and a multiplexor. The latter selects which marking state $M(p_1, \cdots, p_N)$ to consider according to the number of push registers enabled by ctrl, e.g. if 3 stages are enabled, the multiplexor propagates the output of the C-element c_3: $M(p_1, p_2, p_3)$, see grey part in Figure 12a.

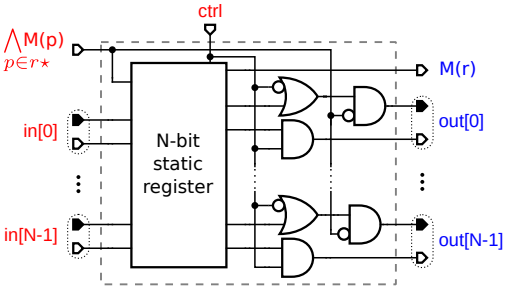
In the approach in Figure 12b, on the other hand, the reconfiguration mechanism is distributed across the implementation of push and pop registers. In this approach, static registers are mapped to the structure in Figure 13a, and push and pop registers are mapped to Figures 13b and 13c, respectively, whose implementation is described in the following. The core of both push and pop registers include a static register. Push registers



(a) N-bit static register.



(b) N-bit push register.

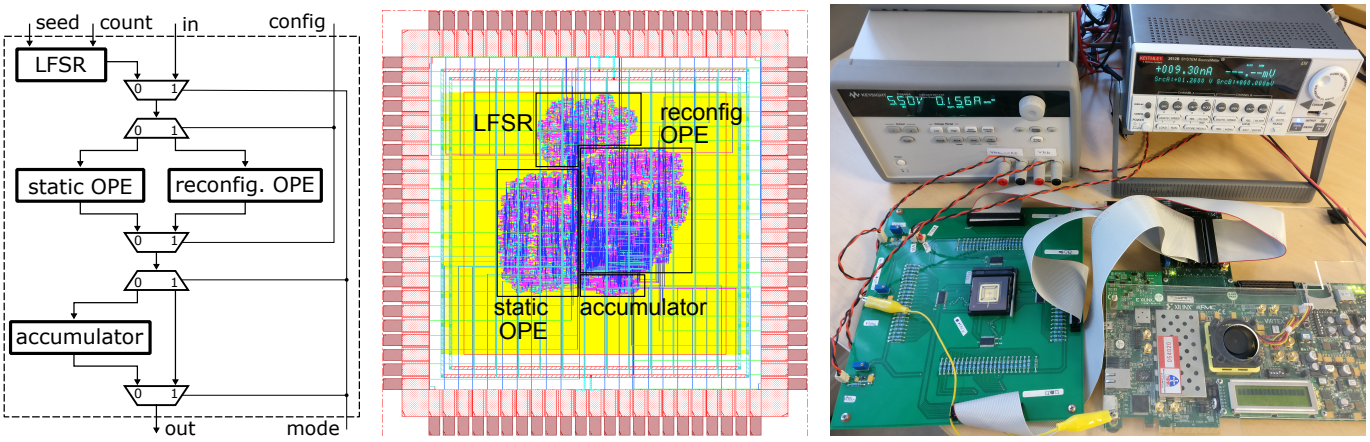


(c) N-bit pop register.

Fig. 13: Static and dynamic register implementations.

are extended with extra-logic that allows them to behave as static registers when ON (ctrl = 1), and to propagate an empty marking state M(r) driven by the R-preset marking state M(q) input when OFF (ctrl = 0), see Figure 13b. Pop registers, in turn, also extends the static register implementation with extra-logic that allows them to behave as static registers when ON, and propagate an empty data value out encoded by a logic 0, driven by their R-postset marking state M(p), when OFF, see Figure 13c. This approach allows the interconnection of r and its fanout registers p via a tree of C-elements for the propagation of the fanout marking state $M(p_1, \cdots, p_N)$, see grey part in Figure 12b. Notice that the implementation of push and pop registers can be customised to accommodate application needs, e.g. pop registers propagating a logic 1 as empty output data value.

Notice that the only constraint on the reconfiguration is the need to keep the ctrl bus stable throughout an OPE run, as the



(a) High-level structure. (b) Floorplan. (c) Testbench setup.

Fig. 14: Ordinal pattern encoding chip.

pipeline depth does not change while data is processed. This was achieved by a single-rail control circuitry that carries the right constants via the ctrl bus, keeping them stable throughout the run. The control registers in Fig. 10 were not synthesised.

The presented back-end library can be further adapted to enable push and pop registers to be reconfigured on the fly. In this case, control tokens should be encoded as dual-rail signals, and would be constrained by the rules of the NCL-D logic, e.g. push registers would only propagate data tokens when both data and control tokens arrive.

The prototyped OPE chip makes use of the reconfigurable interconnection approach to implement all internal reconfiguration scenarios, e.g. final code aggregation. The chip evaluation (see Section IV) highlighted a delay in the computation caused by the daisy chain of C-elements, which led us to design the decentralised reconfigurability approach for avoiding this issue. This latter approach has two advantages: (a) it is more scalable due to the tree of C-elements that has a logarithmic latency with respect to the number of pipeline stages, and due to the lack of the multiplexor. (b) It is more flexible, as push and pop registers can acknowledge empty marking states and data values when disabled. This enables such registers to be disabled in any order without affecting interconnections to their preset and/or postset. The physical implementation and evaluation of the distributed reconfiguration approach is left for future research.

IV. Evaluation

Figure 14a shows the top-level schematic of the designed evaluation chip. It comprises two implementations of the OPE pipeline, *static* and *reconfigurable*, that are activated by the config input. The static implementation is a fixed 18-stage pipeline, and the reconfigurable one supports 16 different depth settings, from 3 to 18 stages. Note that the pipeline depth determines the OPE window size. The reconfigurable OPE (which utilises 124,622 transistors) has an area overhead of ≈28%, compared to the static OPE (97,334 transistors), due to the extra control logic for dynamic reconfiguration.

The chip can be used in *normal* or *random* mode, as selected by the mode input. In the normal mode, an input data stream is supplied via the in port and the results are produced at the out port at every iteration. In the random mode, a series of count random numbers is generated using a linear-feedback shift register (LFSR) based on a user-defined seed. A checksum of the output stream is calculated in the accumulator and a single data item is produced after all generated data is processed. This mode is essential for accurate measurements of the chip performance and energy consumption, as it removes the overheads for interfacing the chip to the testbench environment. The produced checksum is validated against the output of the OPE behavioural model initialised with the same seed and count parameters.

The chip floorplan and its main components are shown in Figure 14b. It was fabricated using TSMC 90nm CMOS technology for low-power applications via Europaractice [30].

A custom test PCB was developed to interface the packaged chip with a Xilinx Virtex 7 FPGA board. A series of experiments was run in the random mode for a stream of 16M LFSR-generated numbers, at supply voltages from 0.3V to 1.6V. Note that such a wide range of voltages was chosen only to exercise the chip in the sub-threshold region, which is of particular interest for QDI circuits. However, much care should be taken before doing this in a production chip as transistors operate reliably only within 10-15% margin of the nominal voltage. The computation time was measured by the FPGA with 1ms precision, the power was monitored using Keithley 2612B System source meter [31], with 1nW accuracy. The testbench setup is shown in Figure 14c.

A. Objectives

The objectives of the experiments are (1) to validate the presented design flow, based on the DFS formalism, on a real-life case study; (2) to evaluate our library of NCL-D style asynchronous components, used to map our DFS model of the OPE to the corresponding digital circuit; and (3) to compare the resulting dynamically reconfigurable and static

implementations, for determining advantages and drawbacks of asynchronous dynamic reconfigurability.

Another solution for accelerating the OPE for a set of window sizes would be to use the static reconfigurability feature of Field Programmable Gate Arrays (FPGA). The latter is different than dynamic reconfigurability, as hardware architecture changes are achieved via total or partial circuit re-synthesis rather than via extra on-board control logic. It is not in the scope of our work to compare dynamic to static reconfigurability. We refer the reader to a paper where ASIC and FPGA characteristics are compared [32].

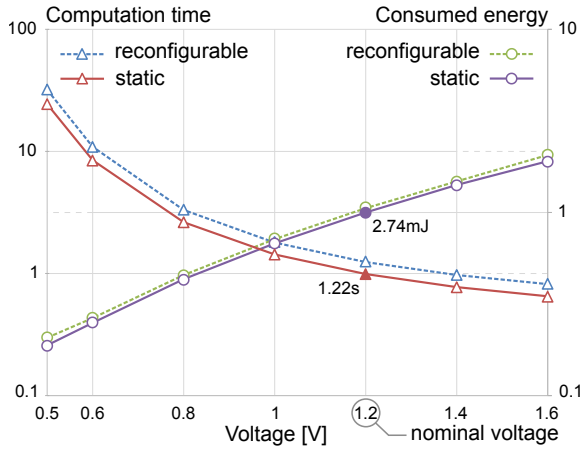
It is also not in the scope of our work to compare our asynchronous accelerator to its synchronous counterpart. Synchronous and asynchronous circuits have been extensively compared in literature under many perspectives – e.g. in terms of power consumption [33], performance [34], resilience [35] – and their advantages and drawbacks are known [8]. Also, the style of implementation that we chose for our design (Null Convention Logic [29]) has been studied and compared to synchronous logic [36].

B. Experiments at varying voltage supply

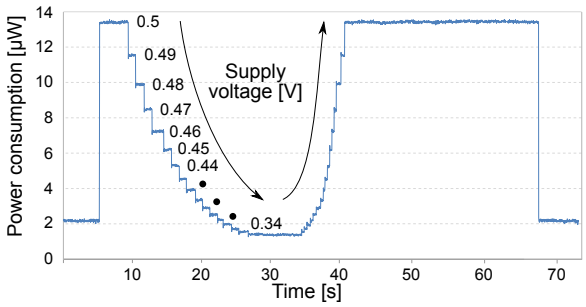
The chip is fully asynchronous and can therefore operate in a wide range of voltages, dynamically adapting its speed. The computation time and energy consumption are characterised in Figure 15a for supply voltages from 0.5V to 1.6V. The length of the reconfigurable pipeline (dashed lines) is set to the maximum value and matches that of the 18-stage static pipeline (solid lines). Both the computation time and the consumed energy are normalised to the corresponding measurements of the static pipeline at the nominal voltage of 1.2V (the reference values are 1.22s and 2.74mJ, respectively). As expected, the lower the voltage the slower, but at the same time more energy-efficient, is the circuit. The energy consumption of the reconfigurable implementation is slightly higher (5% overhead) due to the additional control logic for managing the pipeline configuration. The high computation time of the reconfigurable pipeline (36% overhead) is due to an inefficient implementation of the synchronisation between the stages using a daisy-chain C-element structure. This can be significantly improved (estimates overhead below 10%) using a tree-like C-element structure, see Section III-B.

Note that a C implementation of 18-stage OPE algorithm takes 16s on an Intel-i7 2.30GHz PC (throughput of ≈ 0.06 OPE per second), which is an order of magnitude slower than using the static and reconfigurable pipelined accelerators, which have a throughout of 0.81 and 0.61, respectively.

Another experiment demonstrates the capability of asynchronous pipelines to operate correctly at an unstable voltage supply, down to the near-threshold values. Figure 15b shows the power consumption of the reconfigurable OPE pipeline (with all 18 stages activated) during a single LFSR-generated experiment. At the very beginning (the left side of the graph), the voltage is set to 0.5V, the circuit does nothing, and the power consumption is due to the leakage current. Then, the up spike represents the beginning of the computation.



(a) Computation time and energy consumption at different voltages.



(b) Power consumption at changing supply voltage.

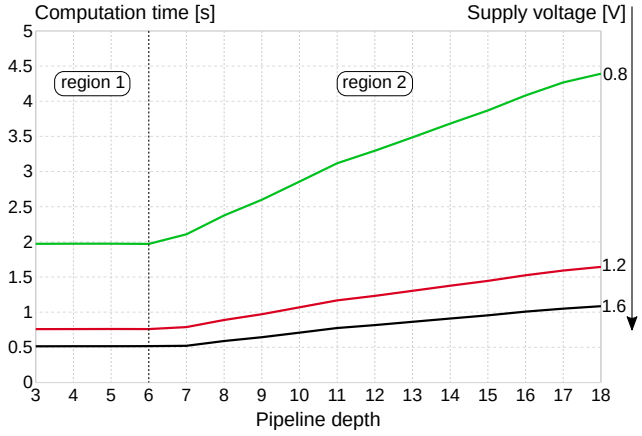
Fig. 15: Experimental results for ordinal pattern encoding chip.

Throughout the experiment, we gradually decreased the supply voltage down to 0.34V (the circuit operated incorrectly at lower voltage). At this voltage the chip operation is frozen – it can be left at this voltage for hours with no progress being made. When the voltage is raised up again the circuit recovers and completes the remaining part of the computation (down spike) correctly.

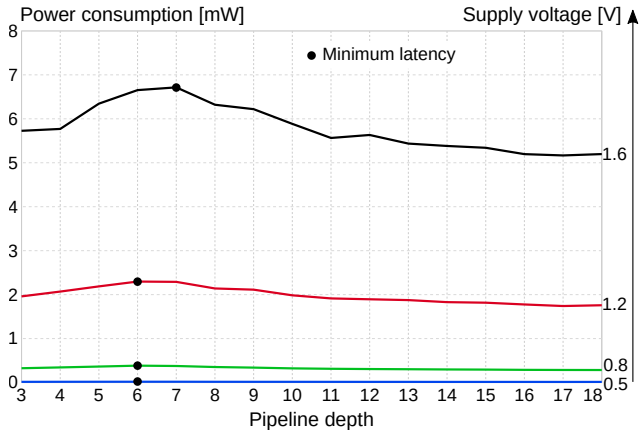
C. Experiments at varying pipeline depth

All configurations of the reconfigurable pipeline (from 3 to 18 stages) were exercised and functionally verified at 0.5-1.6V. Figure 16 describes the behaviour of the OPE chip under varying pipeline depth in terms of computation time, power consumption, and consumed energy. We restrict the analysis to the range of supply voltages $\{0.5, 0.8, 1.2, 1.6\}$ for improving the readability of the shown diagrams.

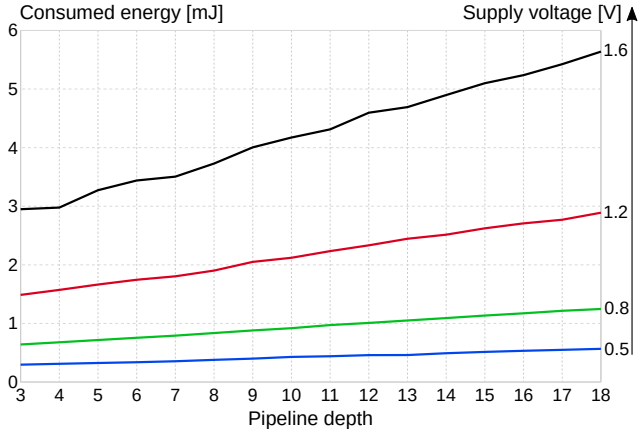
Figure 16a shows the computation time of the chip. When the pipeline depth is lower or equal than 6 stages (region 1), the computation time is limited by the accumulator (see Figure 14a), whose size is fixed by the maximum OPE result size. Reducing the number of stages below 6 does not reduce the computation time. In region 1, the computation time is only scaled down exponentially by the supply voltage reduction. On the other hand, when the pipeline depth is higher than 6 (region 2), the computation time of the chip is limited by



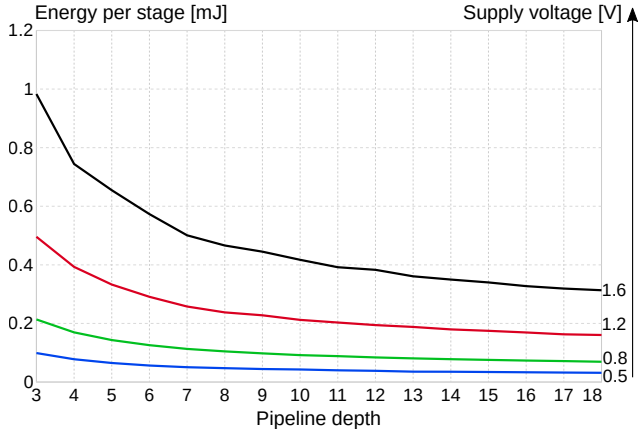
(a) Computation time.



(b) Power consumption.



(c) Energy per computation.



(d) Energy per computation per active stage.

Fig. 16: Evaluation of the reconfigurable OPE pipeline at different voltages (from 0.8V to 1.4V) and pipeline depths.

the synchronisation time between the pipeline stages, whose delay is directly-proportional to the pipeline depth due to the daisy-chain of C-elements, see Section III-B. In this region, the slope of the increment is reverse-proportional with the supply voltage. In this diagram, the experiments corresponding to the supply voltage of 0.5V are omitted for the sake of readability: the computation time is 19.4s in the region 1, and grows up to 42.4s when the pipeline depth is 18.

Figure 16b shows the power consumption of the chip. It increases with the pipeline depth up to the *minimum latency* point (which falls at 6/7 stages depending on the supply voltage), and subsequently decreases. The minimum latency point represents the situation in which the computation time of the accumulator and of the synchronisation time between pipeline stages are balanced, resulting in a higher throughput. In other words, data tokens can move faster within the pipeline, exercising more parts of the chip concurrently, and thus maximising the power consumption. Since the diagram scale does not allow to read the power consumption of the experiments corresponding to 0.5V supply voltage, we report three results from these experiments in the form of (pipe depth, power [μW]) to show that the 0.5V experiments have the same trend: $\{(3, 15.2), \cdots, (6, 17.5), \cdots (18, 13.4)\}$.

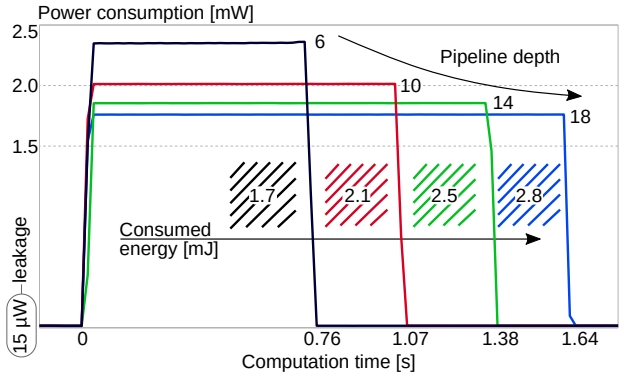


Fig. 17: Time, power, and energy per computation at different pipeline depths at the nominal supply voltage 1.2V.

Figures 16c and *d* show the energy consumption of the chip per computation and per active stage, respectively. The former increases linearly with the depth of the pipeline, the slope of its increment is directly-proportional to the supply-voltage. The latter, in turn, decreases exponentially with the depth of the pipeline, i.e. the amount of work done per energy grows, resulting in a higher chip efficiency. Notice, however, that the

pipeline depth is determined by the workload in applications that make use of OPE, rather than by the energy efficiency.

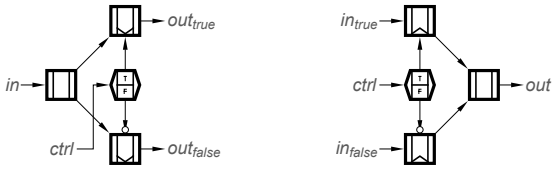
The data described in Figures 16 is summarised in *Figure 17*. It shows the computation time, power consumption and consumed energy at the nominal voltage 1.2V during four LFSR-generated experiments: with the pipeline depth set to 6, 10, 14 and 18. The power consumption is higher when the depth is set to 6 (minimum latency), and decreases with the increment of the pipeline depth. The computation time grows linearly with the increase of the pipeline depth (region 2 of Figure 16a). The consumed energy is shown within the area delimited by the power consumption lines: one can see that it also increases with the pipeline depth. When the chip is waiting for the test to start, the power consumption is due to the leakage current of 15μW.

D. Conclusions

The experiments demonstrate the high degree of flexibility and resilience of the fabricated OPE accelerator: it supports flexible window size (via reconfiguration of the pipeline depth) and can operate at a variable supply voltage (thanks to its asynchronous implementation). The cost of the reconfigurability is 5% in terms of power consumption and 36% in terms of performance (can be improved to 10% in a future prototype).

V. Related work

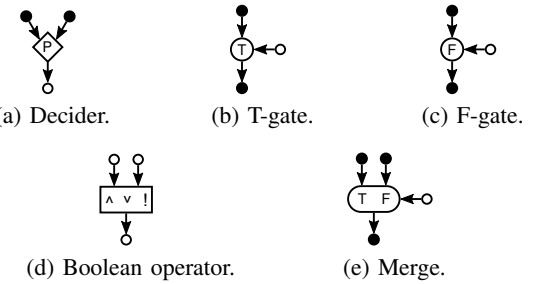
The Dataflow Structures is a *Model of Computation* (MoC), i.e. a set of laws that describes the interaction of the components of a system. Interested readers can find the description of a number of dataflow-based MoCs in [37]. In the next paragraphs, we relate the presented Dataflow Structures formalism to similar existing MoCs that are available in literature.



(a) Demultiplexer. (b) Multiplexer.

Fig. 18: DFS models of a demultiplexer and multiplexer.

In [8], Sparsø and Furber introduced the SDFS formal model, and showed how to use it to describe asynchronous circuits at a high level of abstraction. The model includes logic and register nodes, described in Section II-A, and two extra primitives – *demultiplexer* and *multiplexer* – that are passive to the handshake communication mechanism and are necessary for describing hardware reconfigurability. These two primitives were not taken into consideration when the SDFS was formalised [9] in order to keep the behavioural semantics simple. With the dynamic extension that we propose, demultiplexers and multiplexers can be constructed using static and dynamic registers — see Figure 18. This allows us to keep the semantics of the model relatively simple because the semantics of dynamic registers can be derived from the not too dissimilar semantics of static registers.



(a) Decider. (b) T-gate. (c) F-gate.

(d) Boolean operator. (e) Merge.

Fig. 19: The nodes of the Data flow schema MoC [38].

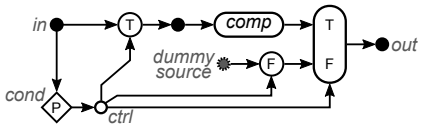
The idea of using Boolean-controlled nodes for describing conditional behaviour in dataflow graphs goes back to 1970s, when Dennis et al. described a MoC for representing both static and dynamic behaviours, namely *Data flow schema* [38]. Similarly to the presented DFS model, data flow schema can handle both data and control types of tokens. Data tokens are used to abstract data values, while control tokens (relying on Boolean conditions) manage the topology, thereby the behaviour, of the model itself at runtime. Figure 19 shows the five nodes of this model as described in the original paper, where • represents data links for the propagation of data tokens, and ○ represents control links for control tokens. The *decider* node takes two data tokens as input, tests an internal predicate P and produces a control token (either True or False). The *T-gate* and *F-gate* nodes behave as the shown push registers: they propagate the input data token to the output only if the input control token is satisfied (i.e. True for the T-gate, and False for the F-gate), otherwise the input data token is consumed from the input link and destroyed. The *Boolean operator* node computes Boolean operations over multiple control tokens. Finally, the *merge* node behaves as a multiplexer, i.e. the input control token selects which of the input data tokens have to be propagated to the output.

(a) Switch. (b) Select.

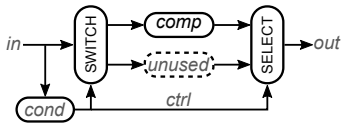
Fig. 20: Control-flow nodes of the BDF MoC [39].

Another formalism based on this idea is the *Boolean-controlled Dataflow* (BDF) [39]. This model also handles both data and control tokens for describing static and conditional behaviours. The BDF extends the *Synchronous Dataflow* (SDF) [40] formalism by introducing two primitives: *switch* and *select*, shown in Figure 20, which practically behave as demultiplexers and multiplexers and enable the representation of conditional behaviour.

Compared to these MoCs, we argue that the set of nodes of the DFS model is *more efficient*. We have shown that the primitives representing conditional computation in existing MoCs, such as the above merge, switch and select, can be decomposed into DFS demultiplexer/multiplexer snippets, see



(a) Data flow schema model.



(b) Boolean-controlled Dataflow model.

Fig. 21: Conditional application of a function comp.

Figure 18, therefore the DFS model is at least as expressive. Furthermore, when modelling reconfiguration scenarios, DFS dynamic nodes provide more flexibility and expressiveness than existing MoCs.

Recall our motivating example where the conditional application of a complex function is concisely modelled with DFS, see Figure 1b. This cannot be naturally expressed using the formalisms in [38], [39] and requires some tricks, as shown in Figure 21. Unlike the DFS model, here a redundant *dummy source* (i.e. which generates redundant data tokens) or *unused* path need to be added in order to allow token propagation when the comp function should be bypassed. Having an efficient and expressive set of nodes is important, as this directly impacts the simplicity of derived models and the characteristics of resulting implementations.

Push and pop registers, as shown in our motivating examples, provide data tokens to their intermediate nodes (e.g. see *comp* in Figure 1b) only when their execution is required, disconnecting them from the rest of the system otherwise. This idea was also applied to asynchronous circuits in [41], and is named *operand isolation*. In the paper, the authors describe two primitives, namely *send* and *receive*, that act as the DFS push and pop registers, respectively, and isolate internal modules when their execution is not required for saving dynamic power. Our work, in comparison, employs this concept at a higher level of abstraction reusing formalisms that are well known to the asynchronous community. In addition, the receive and send primitives are only described behaviourally, whereas, in Section III, we propose a standard-cell based implementation for the push and pop registers that can be employed out-of-the-box in asynchronous circuits functioning with the 4-phase handshake protocol.

VI. Discussion and Future Research

The paper presents the DFS formalism for modelling reconfigurable asynchronous pipelines and defines its behavioural semantics using Petri Nets. The development, verification and synthesis of DFS models are automated in Workcraft. The DFS theory and tools are validated in an ASIC prototype of a reconfigurable OPE accelerator. The chip measurements confirm the expected characteristics of the asynchronous pipeline in terms of performance, energy consumption, and resilience.

Our current tool-chain does the performance analysis at the level of dataflow structures, informing the designer of potential high-level issues, such as insufficient number of data tokens in a critical loop or imbalance between parallel dataflow branches leading to inefficient hardware utilisation. Once the designer is satisfied with high-level dataflow characteristics, it is possible to export the circuit netlist in Verilog format, where conventional tools take over. The error which led to the performance problem identified in the Section IV was made in the conventional part of the flow, outside of the presented tool-chain. To avoid such errors in future, it is important to provide a way to iterate between the high-level performance modelling in Workcraft and the netlist-level modelling in standard EDA tools, where such low-level issues can be revealed.

The DFS formalism abstracts away the implementation details of digital circuits, providing an environment where engineers can focus on the circuit functionality. At present, the elaborated semantics models the 4-phase handshake protocol [29]. The description of additional semantics for other communication protocols (e.g. 2-phase handshake, bundle-data) is left for future research.

Our DFS-based description of the OPE was mapped to a digital circuit relying on the quasi-delay-insensitive timing model [8]. The description can be, nonetheless, used to derive implementations in compliance with other existing timing models (e.g. delay-insensitive, self-timed), by adopting different corresponding backend libraries and flows. The automated generation of design constraints for respecting commonly used timing models (e.g. timing constraints for isochronic forks in QDI) is left for future research.

The future work also includes the development of synthesis backends for popular asynchronous pipeline styles [7], design of a high-level DSL for reconfigurable dataflow graphs, and application of the presented method to large-scale distributed dataflow graphs in the domains of machine learning [2] and application-specific high-performance computing [4]. Thanks to the regularity of circuits for each DFS node type, it might be possible to turn each node into a cell, parameterised by the number of input and output channels. Similar to memory compilers [42], [43] layout of such cells could be fully automated.

Acknowledgements

This research was supported by EPSRC Impact Acceleration grant Dataflow Computation à la Carte, and EPSRC research grants A4A (EP/L025507/1) and POETS (EP/N031768/1).

References

[1] Arvind, D. Culler: *"Dataflow Architectures*. MIT, 1986.

[2] M. Abadi *et al.*: *"TensorFlow: large-scale machine learning on heterogeneous distributed systems"*. White paper, Google Research, 2016.

[3] M. Abadi: *"TensorFlow: learning functions at scale"*. Proc. Int. Conf. on Functional Programming, 2016.

[4] T. Becker, O. Mencer, G. Gaydadjiev: *"Spatial programming with OpenSPL"*. FPGAs for Software Programmers, Springer, pp. 81–95, 2016.

[5] D. Sokolov, V. Dubikhin, V. Khomenko, D. Lloyd, A. Mokhov, A. Yakovlev: *"Benefits of asynchronous control for analog electronics: multiphase buck case study"*. Proc. Design Automation and Test in Europe (DATE), 2017.

[6] S. Chatterjee, M. Kishinevsky, U. Ogras: *"xMAS: quick formal modelling of communication fabrics to enable verification"*. IEEE Design and Test of Computers, v.29(3), pp. 80–88, 2012.

[7] S. Nowick, M. Singh: *"High-performance asynchronous pipelines: an overview"*. IEEE Design & Test of Computers, 28(5), pp. 8–22, 2011.

[8] J. Sparsø, S. Furber: *"Principles of asynchronous circuit design: a systems perspective"*. Kluwer Academic Publishers, 2001.

[9] D. Sokolov, I. Poliakov, A. Yakovlev: *"Analysis of static data flow structures"*. Fundamenta Informaticae, vol. 88(4), pp. 581–610, 2008.

[10] C. Petri: *"Kommunikation mit automaten"*. University of Bonn, PhD thesis, 1962.

[11] D. Sokolov, V. Khomenko, A. Mokhov: *"Workcraft: Ten years later"*. In *"This asynchronous world. Essays dedicated to Alex Yakovlev on the occasion of his 60th birthday"*, Ed. A. Mokhov, pp. 269–293, 2016. http://async.org.uk/ay-festschrift/paper25-Alex-Festschrift.pdf

[12] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, A. Yakovlev: *"Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers"*. IEICE Transactions on Information and Systems, vol. E80-D(3), pp. 315–325, 1997.

[13] V. Khomenko, M. Koutny, A. Yakovlev: *"Logic synthesis for asynchronous circuits based on STG unfoldings and incremental SAT"*. Fundamenta Informaticae, vol. 70(1-2), pp. 49–73, 2006.

[14] S. Burns, A. Martin: *"Syntax-directed translation of concurrent programs into self-timed circuits"*. Proc. Advanced Research in VLSI, pp. 35–50, 1988.

[15] K. van Berkel, J. Kessels, M. Roncken, R. Saeijs, F. Schalij: *"The VLSI-programming language Tangram and its translation into handshake circuits"*. Proc. European Conference on Design Automation, pp. 384–389, 1991.

[16] A. Bardsley, D. Edwards: *"The Balsa asynchronous circuit synthesis system"*. Forum on Design Languages, vol. 224, 2000.

[17] A. Lines: *"Pipelined asynchronous circuits"*, California Institute of Technology, PhD thesis, 1998.

[18] C. Guo, W. Luk, S. Weston: *"Pipelined reconfigurable accelerator for ordinal pattern encoding"*. Proc. Application-specific Systems, Architectures and Processors (ASAP), pp. 194–201, 2014.

[19] D. Sokolov, A. de Gennaro, A. Mokhov: *"Reconfigurable asynchronous pipelines: from formal models to silicon"*. Proc. Design Automation and Test in Europe (DATE), 2018.

[20] A. Mokhov, V. Khomenko: *"Algebra of parameterised graphs"*. ACM Transactions on Embedded Computing Systems, 13(4s), 2014.

[21] L. Rosenblum, A. Yakovlev: *"Signal graphs: from self-timed to timed ones"*. Proc. Int. Workshop on Timed Petri Nets, pp. 199–206, 1985.

[22] I. Poliakov, A. Mokhov, A. Rafiev, D. Sokolov, A. Yakovlev: *"Automated verification of asynchronous circuits using circuit Petri nets"*. Proc. Int. Symp. on Asynchronous Circuits and Systems (ASYNC), pp. 161–170, 2008.

[23] C. Hoare: *"Communicating sequential processes"*, Prentice-Hall, 1985.

[24] J.-R. Abrial: *"Modelling in Event-B"*. Cambridge University Press, 2010.

[25] U. Norell: *"Towards a practical programming language based on dependent type theory"*. Chalmers University of Technology, PhD thesis, 2007.

[26] E. Brady: *"Idris, a general-purpose dependently typed programming language: Design and implementation"*. Journal of Functional Programming, vol. 23(5), pp. 552–593, 2013.

[27] V. Khomenko: *"A usable reachability analyser"*. Technical Report, CS-TR-1140, Newcastle University, 2009.

[28] C. Brej: *"Wagging logic: implicit parallelism extraction using asynchronous methodologies"*. Proc. Application of Concurrency to System Design (ACSD), pp. 35–44, 2010.

[29] K. Fant, S. Brandt: *"Null convention logic: a complete and consistent logic for asynchronous digital circuit synthesis"*. Proc. Application-Specific Systems, Architectures and Processors (ASAP), pp. 261–273, 1996.

[30] Europractice IC website: *"TSMC* $90nm$ *technology overview"*. http://www.europractice-ic.com/technologies_TSMC.php?tech_id=90nm.

[31] Keithley 2612B system source meter data sheet. http://www.testequipmentdepot.com/keithley/pdfs/2600b_datasheet.pdf

[32] I. Kuon, J. Rose: *"Measuring the gap between FPGAs and ASICs"*. IEEE Tran. on Computer-Aided Design of Integrated Circuits and Systems, vol. 26, pp. 203–215, 2007.

[33] L. S. Nielsen, J. Sparsø: *"Designing asynchronous circuits for low power: an IFIR filter bank for a digital hearing aid"*. Proceedings of the IEEE, vol. 87, pp. 268–281, 1999.

[34] A. J. Martin, A. Lines, R. Manohar, M. Nystrom, P. Penzes, R. Southworth, U. Cummings, and T. K. Lee: *"The design of an asynchronous MIPS R3000 microprocessor"*. Proc. Conf. on Advanced Research in VLSI, pp. 164–181, 1997.

[35] L. F. Cristófoli, A. Henglez, J. Benfica, L. Bolzani, F. Vargas, A. Atienza, F. Silva: *"On the comparison of synchronous versus asynchronous circuits under the scope of conducted power-supply noise"*. Proc. Asia-Pacific Int. Sym. on Electromagnetic Compatibility, pp. 1047–1050, 2010.

[36] R. Sovani, K. Haque, and P. Beckett: *"Short word length NULL convention logic FIR filter for low power applications"*. Proc. IEEE Int. WIE Conf. on Electrical and Computer Engineering (WIECON-ECE), pp. 102–105, 2015.

[37] A. Bouakaz, P. Fradet, A. Girault: *"A survey of parametric dataflow models of computation"*. ACM Transactions on Design Automation of Electronic Systems, vol. 22, pp. 38:1–38:25, 2017.

[38] J. Dennis, J. Fosseen, J. Linderman: *"Data flow schemas"*. Massachusetts Institute of Technology, 1972. https://link.springer.com/content/pdf/10.1007/3-540-06720-5_15.pdf

[39] J. Buck: *"Scheduling dynamic dataflow graphs with bounded memory using the token flow model"*. University of California, Berkeley, PhD thesis, 1993.

[40] E. Lee, D. G. Messerschmitt: *"Synchronous data flow"*. Proceedings of the IEEE, vol. 75(9), pp. 1235–1245, 1987.

[41] A. Saifhashemi, P. Beerel: *"Observability conditions and automatic operand-isolation in high-throughput asynchronous pipelines"*. In the book "Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation". Heidelberg, Germany: Springer, 2012, pp. 205–214.

[42] M. Guthaus, J. E. Stine, S. Ataei, B. Chen, B. Wu, M. Sarwar: *"OpenRAM: An open-source memory compiler"*. Proc. IEEE Int. Conf. on Computer Aided Design (ICCAD), 2016.

[43] S. Ataei , R. Manohar: *"AMC: An asynchronous memory compiler"*. Proc. Int. Symp. on Asynchronous Circuits and Systems (ASYNC), 2019.