Reconfigurable Asynchronous Pipelines: from Formal Models to Silicon

Danil Sokolov, Alessandro de Gennaro, Andrey Mokhov
School of Engineering, Newcastle University, UK

***Abstract*—*Dataflow pipelines* are widely used in the design of high-throughput computation systems. Real-life applications often require *dynamically reconfigurable pipelines* to differently process data items or adjust to the current operating mode. Reconfigurable synchronous pipelines are known since 1980s and are well supported by formal models and tools. *Reconfigurable asynchronous pipelines* on the other hand, have neither a formal behavioural model, nor mature EDA support, making them unattractive to industry. This paper presents a model and an open-source tool for the design and verification of reconfigurable asynchronous pipelines, and validates this approach in silicon.**

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]. TensorFlow is supported by Google's hardware *tensor processing units*, that can be combined into distributed computation systems. *Spatial computing* [3] 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 [4]. 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 [5].

Asynchronous non-reconfigurable pipelines have also been extensively studied, e.g. [6] provides an in-depth overview of existing hardware implementation styles, while [7] introduces *Static Dataflow Structures* (SDFS), a formal behavioural model for non-reconfigurable asynchronous pipelines further developed in [8]. As we show in Section II, SDFS cannot adequately model dynamic pipeline reconfiguration.

The main result of this paper is an extension of the SDFS model to a universal formalism of *Dataflow Structures* (DFS) that is capable of capturing both static and dynamically reconfigurable asynchronous pipelines. The semantics of DFS components is expressed using *Petri Nets* thus enabling the reuse of established theory and tools developed by the Petri Net community: DFS software is implemented as a plugin for the WORKCRAFT framework that uses Petri Nets as a common language for integrating a number of backend tools.

We also developed a DFS-based design methodology and a set of generic pipeline stage components for building reconfigurable asynchronous pipelines (released under MIT license). The presented approach was validated with the fabrication of a reconfigurable ASIC accelerator for *ordinal pattern encoding* (OPE) [9]. The chip provides real-life data for analysing benefits and costs of dynamic reconfigurability, and highlighting the resilience of asynchronous pipelines to unpredictable environmental conditions, such as unstable voltage supply.

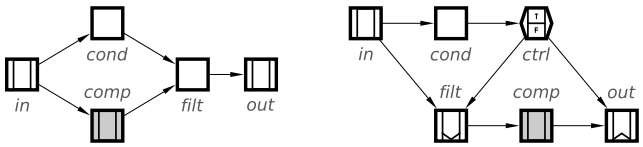
The main contributions of this paper are as follows:

- **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** for DFS modelling (Section II-D), implemented as a plugin for the WORKCRAFT toolset (available at https://www.workcraft.org/).
- **A DFS-based design methodology** for reconfigurable asynchronous pipelines, and its evaluation by the design and verification of OPE pipeline (Section III).
- **An ASIC prototype** implementing both static and dynamically reconfigurable OPE pipelines as a validation of the presented approach (Section IV).

We discuss achieved results and future work in Section V.

II. Dataflow Structures model

As a 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. Fig. 1a shows an SDFS model [7] of such a pipeline, which only supports RTL-style *logic* and *register* components (we show the comp pipeline as a shaded register for simplicity). Note that both cond and comp have to 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*, see Fig. 2.

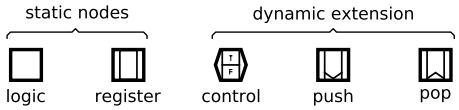


Fig. 2: Five types of DFS nodes.

Fig. 1b shows a DFS model of the motivating example. It applies the cond predicate to the incoming data item, or simply *token*, producing the True or False token in ctrl, which '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.

Formally, a dataflow structure DFS is defined as follows: $\mathsf{DFS} = \langle V, E, M_0 \rangle$, where $V = L \cup R$ is a set of *logic* and *register* nodes, $E \subseteq V \times V$ is the set of *edges* (interconnect), and $M_0 = R \to \{0, 1\}$ is the *initial marking* of registers.

The *preset* and *postset* of a node $x \in V$ are defined as: $\bullet x = \{y : (y, x) \in E\}$ and $x\bullet = \{y : (x, y) \in E\}$.

The *R-preset* and *R-postset* of a node $x \in V$ are defined as: $\star x = \{y \in R : \exists \delta\,(y, x)\}$ and $x\star = \{y \in R : \exists \delta\,(x, y)\}$, where a *logic path* $\delta\,(s, t)$ from $s \in V$ to $t \in V$ 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 $0 < i < n$.

A. Static nodes

A logic DFS node models a combinational dataflow component [7]. It can be *evaluated* when the registers in its preset are *marked* (i.e. contain valid data) and the logic nodes in its preset are evaluated. Symmetrically, a logic node can be *reset* when the registers in its preset are *unmarked* (i.e. contain no data) and the logic nodes in its preset are reset. 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 $Q = S \vee \overline{R} \wedge Q$):

$$\begin{aligned} C\,(l) &= C_\uparrow\,(l) \vee \overline{C_\downarrow\,(l)} \wedge C\,(l) \\ C_\uparrow\,(l) &= \bigwedge_{k \in \bullet l \cap L} C\,(k) \wedge \bigwedge_{q \in \bullet l \cap R} M\,(q) \\ C_\downarrow\,(l) &= \bigwedge_{k \in \bullet l \cap L} \overline{C\,(k)} \wedge \bigwedge_{q \in \bullet l \cap R} \overline{M\,(q)} \end{aligned} \tag{1}$$

A register node models a sequential dataflow component [7]. 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 therefore defined as follows:

$$\begin{aligned} M\,(r) &= M_\uparrow\,(r) \vee \overline{M_\downarrow\,(r)} \wedge M\,(r) \\ M_\uparrow\,(r) &= \bigwedge_{k \in \bullet r \cap L} C\,(k) \wedge \bigwedge_{q \in \star r} M\,(q) \wedge \bigwedge_{q \in r \star} \overline{M\,(q)} \\ M_\downarrow\,(r) &= \bigwedge_{k \in \bullet r \cap L} \overline{C\,(k)} \wedge \bigwedge_{q \in \star r} \overline{M\,(q)} \wedge \bigwedge_{q \in r \star} M\,(q) \end{aligned} \tag{2}$$

B. Dynamic extension

The DFS model introduces *control*, *push*, and *pop* types of register nodes: $R_{ctrl} \cup R_{push} \cup R_{pop} \subseteq R$.

Control registers can only contain True or False. A node is called *true-controlled* or *false-controlled* if all control registers in its R-preset hold the True or False token, respectively. In case of a mismatch, i.e. when both True and False tokens are present in the R-preset, the node is disabled, which may lead to a deadlock. The reachability of such problematic states needs to be formally verified and has been automated.

Both push and pop behave as static registers when true-controlled. A false-controlled push on the other hand consumes and destroys an incoming token, while a false-controlled pop produces an 'empty' token.

With the introduction of dynamic registers the set and reset functions (1) and (2) are refined as shown in (3) and (4), respectively (the superscript 'd' stands for 'dynamic').

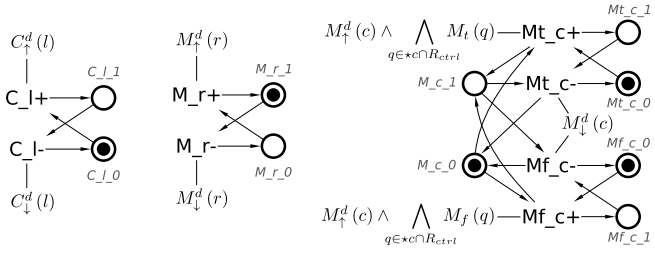
$$\begin{aligned} C^d_\uparrow\,(l) &= C_\uparrow\,(l) \wedge \bigwedge_{p \in \bullet l \cap R_{push}} M_t\,(p) \\ C^d_\downarrow\,(l) &= C_\downarrow\,(l) \wedge \bigwedge_{p \in \bullet l \cap R_{push}} \overline{M_t\,(p)} \end{aligned} \tag{3}$$

$$\begin{aligned} M^d_\uparrow\,(r) &= M_\uparrow\,(r) \wedge \bigwedge_{p \in \star r \cap R_{push}} M_t\,(p) \wedge \bigwedge_{p \in r \star \cap R_{pop}} \overline{M_t\,(p)} \\ M^d_\downarrow\,(r) &= M_\downarrow\,(r) \wedge \bigwedge_{p \in \star r \cap R_{push}} \overline{M_t\,(p)} \wedge \bigwedge_{p \in r \star \cap R_{pop}} M_t\,(p) \end{aligned} \tag{4}$$

The function $M_t\,(p)$ determines if a marked push or pop node $p \in R_{push} \cup R_{pop}$ received a token being true-controlled (i.e. it operates as a static register). The behaviour of dynamic register nodes is defined analogously; for example, a control register node marking $M\,(c)$ is defined as a non-deterministic choice between being marked with the True token $M_t\,(c)$ or the False token $M_f\,(c)$:

$$\begin{aligned} M\,(c) &= M_t\,(c) \vee M_f\,(c)\,,\ c \in R_{ctrl} \\ M_t\,(c) &= M^d_\uparrow\,(c) \wedge \bigwedge_{q \in \star c \cap R_{ctrl}} M_t\,(q) \vee \overline{M^d_\downarrow\,(c)} \wedge M_t\,(c) \\ M_f\,(c) &= M^d_\uparrow\,(c) \wedge \bigwedge_{q \in \star c \cap R_{ctrl}} M_f\,(q) \vee \overline{M^d_\downarrow\,(c)} \wedge M_f\,(c) \end{aligned} \tag{5}$$

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



(a) Logic. (b) Register. (c) Control.

Fig. 3: Petri Net semantics of DFS nodes.

C. Petri Net semantics

We express the execution semantics of DFS components using *Petri Nets* (PN) with the read-arcs extension [10]. The PN semantics enables the reuse of established theory and tools for the formal verification. Other general-purpose formalisms can be used to capture DFS semantics, e.g. finite state machines, process algebra [12] or Event-B [13].

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 $r \in R$. 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 → x+ → x_1 and x_1 → x- → 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 (3) and (4). Fig. 3a shows a PN snippet for a logic node in the reset state and Fig. 3b – for a marked register.

For a control register $c \in R_{ctrl}$ two additional variables Mt_c and Mf_c are used, representing its True and False marking, as shown in Fig. 3c. Note that transition M_c+ is refined into a pair of mutually-exclusive transitions Mt_c+ and Mf_c+. Similarly, M_c- is refined into Mt_c- and Mf_c-. PN translation of push and pop registers is analogous.

The DFS model of our motivating example (Fig. 1b), is translated into a PN shown in Fig. 4. 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. 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 [11]. MPSAT enables verifying DFS models for the standard properties (such as deadlock) and custom functional properties (such as hazards) expressed in Reach language [14].

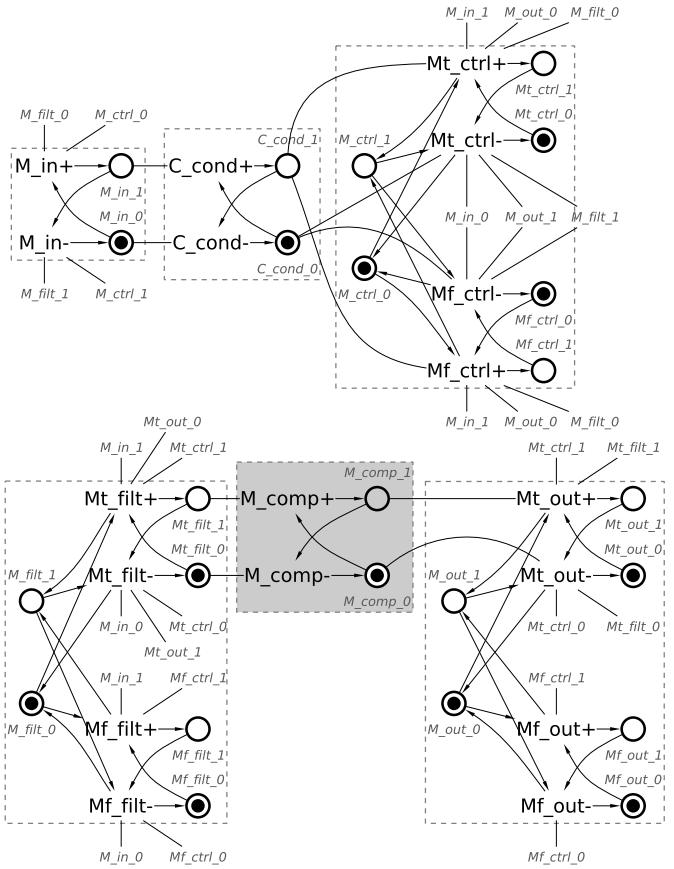


Fig. 4: Petri Net semantics of the DFS in Fig. 1b.

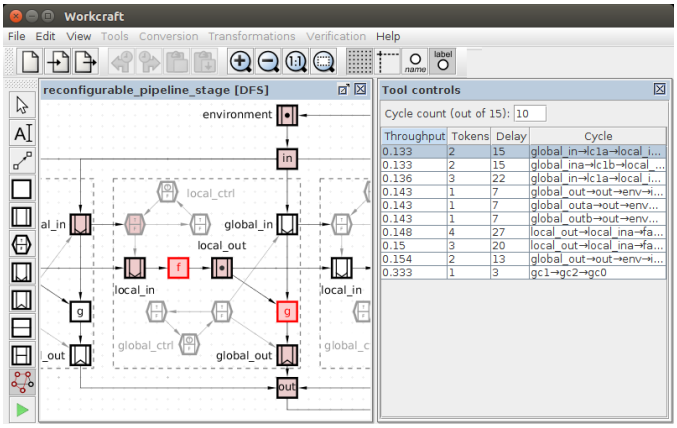


Fig. 5: Performance analysis of DFS in WORKCRAFT.

A screenshot in Fig. 5 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 [15].

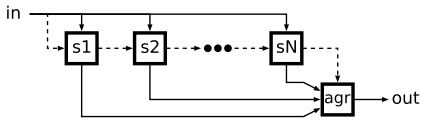
A verified and optimised DFS model can be automatically translated into an asynchronous circuit netlist by directly mapping its nodes into pre-built components and connecting them

according to the dataflow arcs. A library of such components determines the circuit implementation style – see [6] for a comprehensive overview. The circuit is subsequently exported as a Verilog netlist to be used in a conventional backend flow.

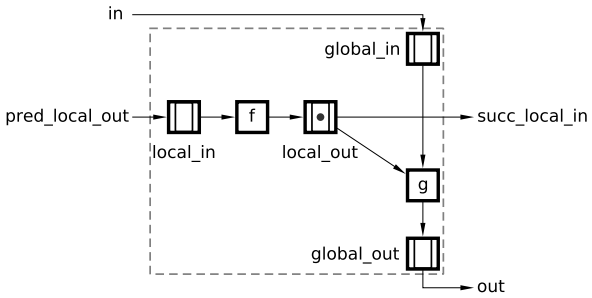
III. Reconfigurable pipelines

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

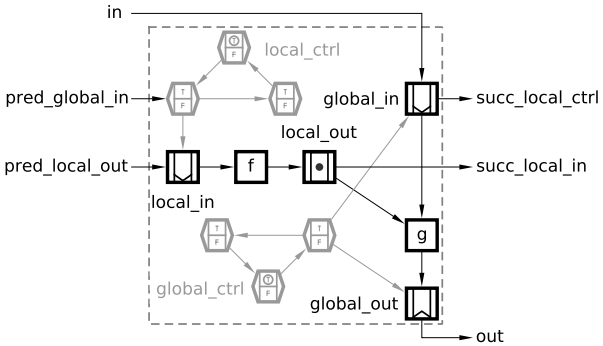
Fig. 6a shows a generic pipeline structure comprising N stages, who interface to each other via *local channels* (dashed arcs), and are connected to the common input in and aggregated output out via *global channels* (solid arcs).



(a) Generic N-stage pipeline.



(b) Static pipeline stage.



(c) Reconfigurable pipeline stage.

Fig. 6: Pipeline with local and global stage interfaces.

A DFS model for a pipeline stage is shown in Fig. 6b. It applies a function f to a token in the local_in register (input data from the previous stage) and produces a token in the local_out register (output data for the next stage). The produced token, paired with the common input token in global_in, 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. Fig. 6c shows our DFS design of a reconfigurable pipeline stage. The local_in interface is implemented as a push register 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.

A. Case study

We apply the proposed methodology to the design of an asynchronous dataflow accelerator for reconfigurable ordinal pattern encoding (OPE) [9]. It 'ranks' the last N items in an incoming data stream[1], 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 $N = 6$. The table below shows the windows starting at different indices within the stream and the corresponding rank lists:

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

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 [9].

Fig. 7 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

[1]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 $(3, 1, 2, 4)$.

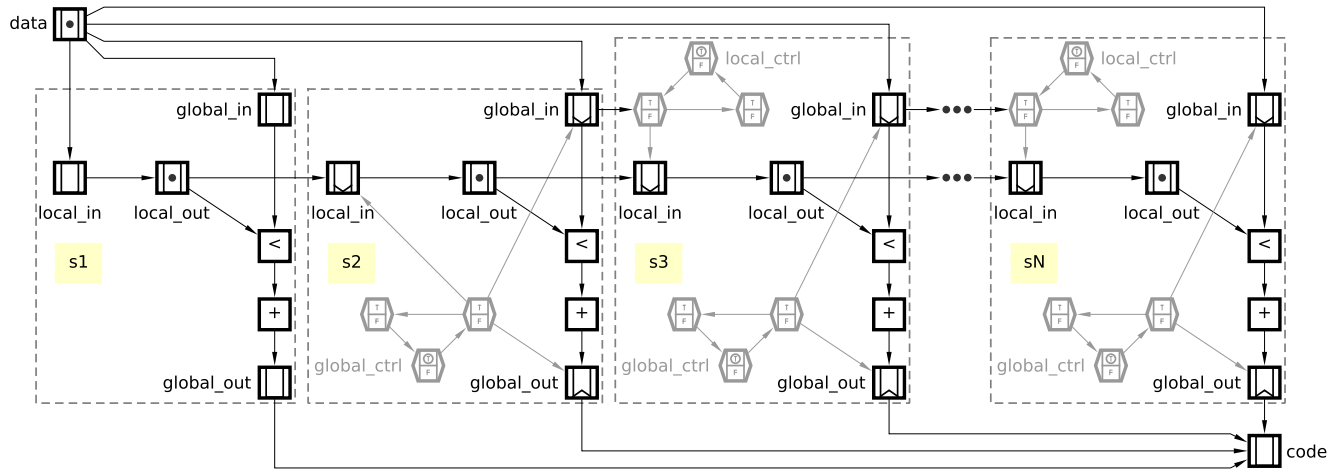


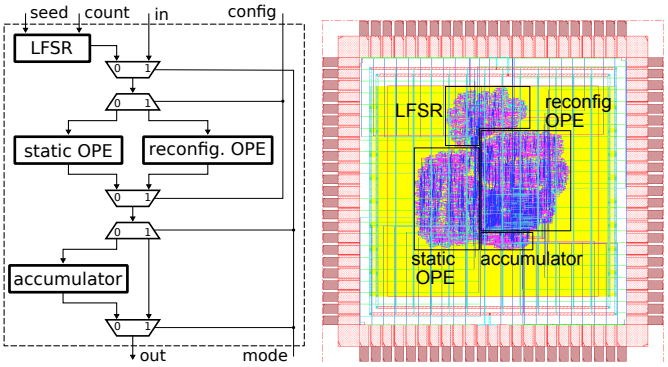
Fig. 7: DFS model of OPE pipeline with reconfigurable depth (from 1 to N stages) that corresponds to the OPE window size.

to incorrect initialisation of control registers) were identified, analysed and corrected during the design process.

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 [16]. A conventional flow was subsequently employed for technology mapping, layout, and place-and-route tasks.

IV. Evaluation

Fig. 8a 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 simple 18-stage pipeline, and the reconfigurable one supports 16 different depth settings, from 3 to 18 stages. Note that the pipeline depth corresponds to the OPE window size.



(a) High-level structure. (b) Floorplan.

Fig. 8: Ordinal pattern encoding chip.

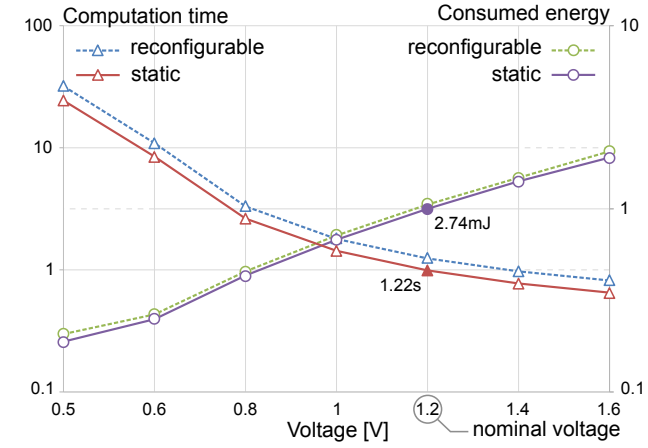
The chip can be used in *normal* or *random* mode, as selected by the mode input. In the normal mode, an input data stream is supplied via the in port and the results are produced at the out port at every iteration. In the random mode, a series of count random numbers is generated using a linear-feedback 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 Fig. 8b. It was fabricated using TSMC 90nm CMOS technology for low-power applications via Europaractice.

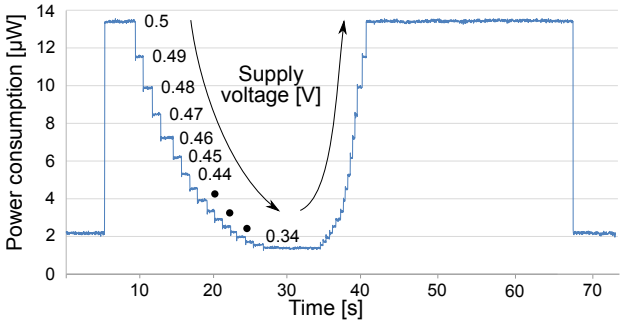
A custom test PCB was developed to interface the packaged chip with a Xilinx Virtex 7 FPGA board. A series of experiments was run in the random mode for a stream of 16M LFSR-generated numbers, at supply voltages from 0.3V to 1.6V. The computation time was measured by the FPGA with 1ms precision, the power was monitored using Keithley 2612B System source meter, with 1nW accuracy.

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 Fig. 9a 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 (as in the static OPE pipeline).

All configurations of the reconfigurable pipeline (from 3 to 18 stages) were exercised at 0.5-1.6V. The experiments showed that both the computation time and the energy consumption in-



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



(b) Power consumption at changing supply voltage.

Fig. 9: Experimental results for ordinal pattern encoding chip.

crease linearly with the pipeline length; the slope of increment is reverse-proportional to the supply voltage.

Another experiment demonstrates the capability of asynchronous pipelines to operate correctly at an unstable voltage supply, down to the near-threshold values. Fig. 9b 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. 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.

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. 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.

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 future work also includes the development of synthesis backends for popular asynchronous pipeline styles [6], 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 [3].

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] T. Becker, O. Mencer, G. Gaydadjiev: *"Spatial programming with OpenSPL"*. FPGAs for Software Programmers, Springer, 2016.
[4] D. Sokolov *et al.*: *"Benefits of asynchronous control for analog electronics: multiphase buck case study"*. DATE, 2008.
[5] 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.
[6] S. Nowick, M. Singh: *"High-performance asynchronous pipelines: an overview"*. IEEE Design & Test of Computers, 28(5), pp. 8–22, 2011.
[7] J. Sparsø, S. Furber: *"Principles of asynchronous circuit design: a systems perspective"*. Kluwer Academic Publishers, 2001.
[8] D. Sokolov, I. Poliakov, A. Yakovlev: *"Analysis of static data flow structures"*. Fundamenta Informaticae, vol. 88(4), pp. 581–610, 2008.
[9] C. Guo, W. Luk, S. Weston: *"Pipelined reconfigurable accelerator for ordinal pattern encoding"*. ASAP, pp. 194–201, 2014.
[10] L. Rosenblum, A. Yakovlev: *"Signal graphs: from self-timed to timed ones"*. International Workshop on Timed Petri Nets, pp. 199–206, 1985.
[11] 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.
[12] C. Hoare: *"Communicating Sequential Processes"*, Prentice-Hall, 1985.
[13] J.-R. Abrial: *"Modelling in Event-B"*. Cambridge Univ. Press, 2010.
[14] V. Khomenko: *"A usable reachability analyser"*. Technical Report, CS-TR-1140, Newcastle University, 2009.
[15] C. Brej: *"Wagging logic: implicit parallelism extraction using asynchronous methodologies"*. ACSD, pp. 35–44, 2010.
[16] K. Fant, S. Brandt: *"Null convention logic: a complete and consistent logic for asynchronous digital circuit synthesis"*. ASAP, 1996.