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 \( \text{comp} \) only to those data items that satisfy an easily-checked condition \( \text{cond} \), 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 \( \text{comp} \) pipeline as a shaded register for simplicity). Note that both \( \text{cond} \) and \( \text{comp} \) have to be executed before filtering unneeded results with filter 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.
The DFS formalism that uses three new types of registers, namely control, push, and pop, see Fig. 2.

Formally, a dataflow structure DFS is defined as follows:

DFS = (V, E, M₀), where V = L ∪ R is a set of logic and register nodes, E ⊆ V × V is the set of edges (interconnect), and M₀ = R → {0, 1} is the initial marking of registers.

The preset and postset of a node x ∈ V are defined as:

\[
\bullet x = \{ y : (y, x) ∈ E \} \quad \text{and} \quad \bullet \bullet x = \{ y : (x, y) ∈ E \}.
\]

The R-preset and R-postset of a node x ∈ V are defined as:

\[
\bullet x = \{ y ∈ R : ∃ δ(y, x) \} \quad \text{and} \quad \bullet \bullet x = \{ y ∈ R : ∃ δ(x, y) \},
\]

where a logic path δ(s, t) from s ∈ V to t ∈ V is a non-empty sequence of edges ((zi−1, zi) ∈ E, i ∈ [1..n]) such that z₀ = s, zₙ = t, and zi ∈ L for 0 < i < n.

A. Static nodes

A logic DFS model needs 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 ∈ L can be defined using the evaluate C⁺(l) and reset C⁻(l) functions (similar to the set/reset functions in the SR latch equation Q = S ∨ R ∧ Q):

\[
\begin{align*}
C(l) &= C⁺(l) \lor C⁻(l) \land C(l) \\
C⁺(l) &= \bigwedge_{k \in l \cap L} C(k) \land \bigwedge_{q \in l \cap R} M(q) \\
C⁻(l) &= \bigwedge_{k \in l \cap L} \exists \bigwedge_{q \in l \cap R} M(q)
\end{align*}
\]

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 ∈ R is therefore defined as follows:

\[
\begin{align*}
M(r) &= M⁺(r) \lor M⁻(r) \land M(r) \\
M⁺(r) &= \bigwedge_{k \in l \cap L} C(k) \land \bigwedge_{q \in l \cap R} M(q) \land \bigwedge_{q \in l \cap R} M(q) \\
M⁻(r) &= \bigwedge_{k \in l \cap L} C(k') \land \bigwedge_{q \in l \cap R} M(q) \land \bigwedge_{q \in l \cap R} M(q)
\end{align*}
\]

B. Dynamic extension

The DFS model introduces control, push, and pop types of register nodes: Rctrl ∪ Rpush ∪ Rpop ⊆ 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’).

\[
\begin{align*}
C⁺(l) &= C⁺(l) \land \bigwedge_{p \in l \cap R_{push}} M(p) \\
C⁻(l) &= C⁻(l) \land \bigwedge_{p \in l \cap R_{push}} M(p) \\
M⁺(r) &= M⁺(r) \land \bigwedge_{p \in l \cap R_{push}} M(p) \land \bigwedge_{p \in l \cap R_{pop}} M(p) \\
M⁻(r) &= M⁻(r) \land \bigwedge_{p \in l \cap R_{push}} M(p) \land \bigwedge_{p \in l \cap R_{pop}} M(p)
\end{align*}
\]

The function M(p) determines if a marked push or pop node p ∈ Rpush ∪ Rpop 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(c) or the False token M(c):

\[
\begin{align*}
M(c) &= M⁺(c) \lor M⁻(c), c ∈ Rctrl \\
M⁺(c) &= M⁺(c) \land \bigwedge_{q \in l \cap R_{ctrl}} M(q) \land \bigwedge_{q \in l \cap R_{ctrl}} M(q) \\
M⁻(c) &= M⁻(c) \land \bigwedge_{q \in l \cap R_{ctrl}} M(q) \land \bigwedge_{q \in l \cap R_{ctrl}} M(q)
\end{align*}
\]
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 $x_0 \rightarrow x+ \rightarrow x_1$ 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 (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 $c \in R_{\text{ctrl}}$ two additional variables $M_t_c$ and $M_f_c$ 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 $M_t_c+$ and $M_f_c+$. Similarly, $M_c-$ is refined into $M_t_c-$ and $M_f_c-$. 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 $M_t_{\text{ctrl}+}$ and $M_f_{\text{ctrl}+}$ can be enabled simultaneously, thus representing a non-deterministic choice for the evaluation result of the $\text{cond}$ logic. The choice between $M_t_{\text{filt}+}$ and $M_f_{\text{filt}+}$ on the other hand is determined by $M_t_{\text{ctrl}+}$ and $M_f_{\text{ctrl}+}$ 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].
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 $\text{in}$ and aggregated output $\text{out}$ via global channels (solid arcs).

![Diagram](image)

(a) Generic $N$-stage pipeline.

ise a Verilog netlist to be used in a conventional backend flow.

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 $\text{local}_\text{in}$ register (input data from the previous stage) and produces a token in the $\text{local}_\text{out}$ register (output data for the next stage). The produced token, paired with the common input token in $\text{global}_\text{in}$, are passed to a function $g$, which produces a $\text{global}_\text{out}$ 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 $\text{local}_\text{in}$ interface is implemented as a push register controlled by the $\text{local}_\text{ctrl}$ structure. The $\text{global}_\text{in}$ and $\text{global}_\text{out}$ are push and pop registers, respectively, controlled by the $\text{global}_\text{ctrl}$ structure. Both $\text{local}_\text{ctrl}$ and $\text{global}_\text{ctrl}$ 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 $\text{local}_\text{ctrl}$ only if the previous stage is included in the pipeline and $\text{global}_\text{in}$ 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:

<table>
<thead>
<tr>
<th>Index</th>
<th>Window</th>
<th>Rank list</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$(3, 4, 1, 5, 9)$</td>
<td>$(3, 1, 4, 2, 5, 6)$</td>
</tr>
<tr>
<td>2</td>
<td>$(1, 4, 1, 5, 9, 2)$</td>
<td>$(1, 4, 2, 5, 6, 3)$</td>
</tr>
<tr>
<td>3</td>
<td>$(4, 1, 5, 9, 2, 6)$</td>
<td>$(3, 1, 4, 6, 2, 5)$</td>
</tr>
</tbody>
</table>

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 $\text{global}_\text{ctrl}$ to control both the local and global interfaces, which is possible because $s1$ is always included in the pipeline and its $\text{global}_\text{in}$ 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 $(3, 1, 2, 4)$. 

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

Design, Automation And Test in Europe (DATE 2018)
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.

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-
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