CEUR-WS.org/Vol-4106/short6.pdf

Dataflow-based FPGA Accelerators Generation via MLIR and Multi-dataflow Composer tool

Mohammad Cheshfar[1,*,†], Jiahong Bi[2,†] and Francesco Ratto[1,*,†]

[1]Department of Electrical and Electronic Engineering, University of Cagliari, 09123, Cagliari, Italy
[2]Faculty of Computer Science, Dresden University of Technology, Dresden, Germany

Abstract
The Multi-Level Intermediate Representation (MLIR) framework is emerging as promising solution to develop extensible and reusable compiler infrastructure for heterogenous hardware. This work presents an automated compilation flow for generating synthesizable FPGA accelerators from high-level dataflow specifications using the MLIR framework and the Multi-Dataflow Composer (MDC) tool. The flow leverages the *dfg-mlir* dialect to model static dataflow graphs and integrates a dedicated lowering pipeline to generate MDC-compatible project files. Actor implementations are synthesized automatically using the *emitHLS* dialect which can target the Vitis HLS tool, enabling compatibility with FPGA toolchains without manual Hardware Description Language (HDL) coding. To validate the proposed automated approach, a comparison with manual HDL-based flow is carried out. FPGA accelerators are synthesized targeting the AMD Kria KV260 platform. Resource usage results confirm functional equivalence and comparable resource utilization and computation time. The integration with the MDC tool opens up the possibility of generating coarse-grained reconfigurable accelerators, which will be explored in future work.

Keywords
Dataflow, MLIR, MDC, High-level synthesis, FPGA

1. Introduction

The MYRTUS project [1] aims to address the growing complexity of designing and deploying applications for heterogeneous computing systems across the edge–cloud continuum. One of its objective is to provide a reference Design and Programming Environment (DPE) for continuum computing systems, featuring interoperable support for cross-layer modelling, threat analysis, application modelling, and code generation.

A central component of this vision is the development of a robust compiler infrastructure capable of supporting a wide range of input programming models and hardware targets. To achieve this, MYRTUS adopts the MLIR framework as one of the core technologies. MLIR's extensible and modular architecture allows the DPE to act as a common integration layer between diverse source languages and backend platforms, supporting both general-purpose and domain-specific compilation pipelines. This choice ensures long-term flexibility and scalability in supporting new devices, programming abstractions, and optimization strategies across the continuum.

Among the tools integrated into the MYRTUS DPE is MDC [2], an open-source design tool for the generation and management of coarse-grain reconfigurable hardware accelerators. MDC provides a high-level environment for creating dataflow-oriented hardware architectures that can be synthesized onto reconfigurable platforms, particularly FPGAs. Its coarse-grained reconfigurable (CGR) approach enables resource sharing among the different accelerators configurations.

However, the current MDC toolchain faces several limitations. First, it relies on the ORCC [3] intermediate representation, which requires dedicated front-ends for each input format, such as QONNX2MDC

CPSWS'25: CPS Summer School PhD Workshop, September 22, 2025, Alghero, Sardinia, Italy
*Corresponding author.
†These authors contributed equally.
mohammad.cheshfar@unica.it (M. Cheshfar); jiahong.bi@tu-dresden.de (J. Bi); francesco.ratto@unica.it (F. Ratto)
https://cfaed.tu-dresden.de/ccc-staff/jiahong-bi (J. Bi); https://web.unica.it/unica/page/it/francesco_ratto (F. Ratto)
0000-0003-1464-0773 (M. Cheshfar); 0009-0002-5827-1148 (J. Bi); 0000-0001-5756-5879 (F. Ratto)
© 2025 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2025-11-25

[4], limiting extensibility. In this work, we address this limitation by introducing an MLIR-based frontend, which replaces the need for multiple format-specific frontends with a unified and extensible IR framework. This enables MDC to leverage existing MLIR importers, reducing engineering overhead and improving interoperability with other MLIR-based tools.

Finally, features central to MDC—such as coarse-grained reconfigurability and dataflow merging—are not natively supported in MLIR. Our MLIR-compatible extension exposes these capabilities, enriching the ecosystem with CGR-style resource-sharing abstractions for reuse across MLIR-based compilers.

The rest of the paper is organized as follows: Section 2 provides an overview of the *dfg-mlir* dialect and the MDC toolchain, outlining the foundational concepts relevant to this work. Section 3 highlights the key contributions and improvements introduced in this study, supported by both simulation and synthesis of a representative example. Finally, Section 4 presents concluding remarks and discusses future research directions aimed at enhancing the proposed methodology and extending its applicability.

2. Background

This section provides a brief overview of the foundational tools and models that underpin the proposed methodology. We present an introduction to the MLIR framework and the *dfg-mlir* [5] dialect developed in previous research projects [1], which serve as the basis for our compilation and intermediate representation. These tools are critical in enabling a structured and extensible transformation flow from a high-level domain-specific language (DSL) to low-level hardware targets [6]. Then, we introduce the MDC, which offers a design suite for coarse-grained reconfigurable accelerators implementation.

2.1. MLIR and the dfg-mlir Dialect

MLIR is a flexible compiler infrastructure developed to support multiple levels of abstraction in modern compiler toolchains. It provides mechanisms for defining custom dialects, allowing domain-specific operations and types to seamlessly integrate into a unified compilation framework [7]. This design enables the representation and progressive lowering of high-level DSLs into hardware-specific implementations.

In the context of dataflow programming, the *dfg-mlir* dialect was introduced as part of the EVEREST project to model applications as dataflow graphs [8], and the MYRTUS project continues the development effort by extending the dialect and enabling deployment across the computing continuum [9]. It supports the specification of Homogeneous Synchronous Dataflow (HSDF) actors through operations such as *dfg.operator*, where each actor is annotated with explicitly defined input and output ports. These ports are implicitly connected via First In First Out (FIFO) channels, allowing asynchronous communication between actors. For more expressive dataflow models, such as Kahn Process Networks (KPNs), the dialect includes a *dfg.process* construct that allows multiple input and output ports per actor [8]. The *convert-to-emitHLS* pass is provided to convert multi-port operators into *dfg.process* operations, maintaining the semantic flexibility of the original graph while enabling analysis and transformation within the MLIR infrastructure.

At the core of this representation lies the *dfg.region*, which serves as the top-level container for the dataflow graph. Within a region, operators are instantiated using the *dfg.operator* construct and interconnected through explicitly typed channels that encode communication and preserve the streaming semantics of data tokens. This modular and high-level design enables early-stage analysis, transformation, and reuse, supporting systematic mapping onto hardware. While the conceptual foundation is introduced here, a concrete illustrative example is deferred to Section 3.2, where it complements the step-by-step methodology and toolchain integration process. This separation allows the example to be presented in the context of the complete compilation flow, enhancing clarity and continuity.

The *dfg-mlir* dialect facilitates integration with hardware backends by enabling passes that analyze and lower dataflow graphs to lower-level representations, paving the way for hardware code generation and deployment on reconfigurable platforms. Its modularity and compatibility with other MLIR dialects make it a suitable foundation for custom FPGA toolchains.

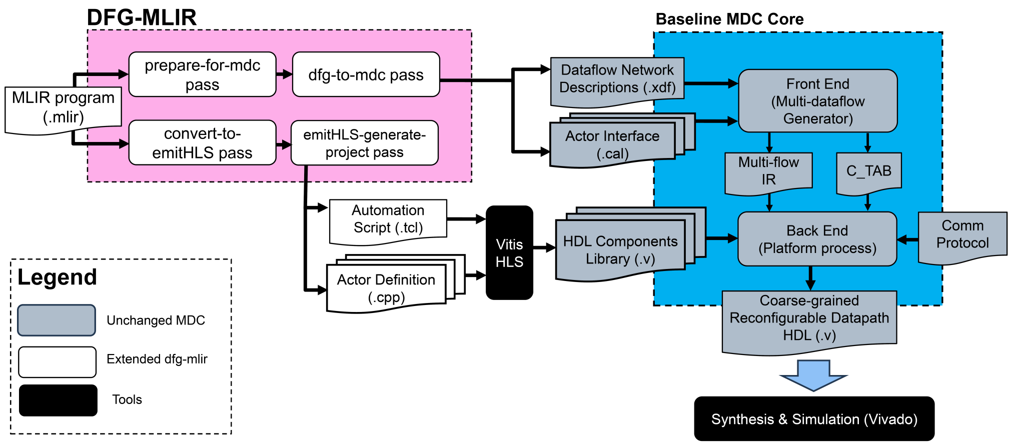


Figure 1: Proposed work: dfg-mlir extension to target coarse grained reconfigurable accelerator through MDC tool.

2.2. Multi-Dataflow Composer

The MDC is an open-source[1] tool suite for automating the design of CGR substrates and supporting their integration in processor-accelerator system environments [10]. MDC enables the generation of synthesizable hardware accelerators starting from dataflow specifications of individual application kernels.

As illustrated in Figure 1, Baseline MDC Core is composed of a front-end and a back-end. The front-end, known as the Multi-dataflow Generator, performs datapath merging—a technique that identifies and combines shared operations across multiple dataflow applications to produce a multi-dataflow reconfigurable network [11]. The resulting architecture contains SBoxes (switch boxes) and Configuration Table,which tracks system programmability to enable time-multiplexed reuse of Processing Elements (PEs). The back-end, known as the Platform Composer, translates the merged dataflow network into a complete SystemVerilog hardware description [12]. Each actor is mapped to a dedicated PE, and data channels are implemented using FIFO buffers. The tool generates the structural elements of the CGR datapath, including control signals, and memory interfaces, resulting in a synthesizable hardware description.

MDC requires the following inputs: (i) dataflow network descriptions, commonly in the *XML Dataflow Format (XDF) format*; (ii) actor implementations in a pre-existing HDL library; and (iii) a communication protocol specification. An Intermediate Representation is produced starting from the *.XDF graphs*, which define the topology of the dataflow network, and associated *.Caltrop Actor Language (CAL) actors*, which describe actor metadata such as port interfaces and data types [10]. From IRs corresponding to those of the input dataflow network descriptions, the tool applies datapath merging and hardware composition algorithms to produce an efficient, time-multiplexed CGR datapath.

3. Work-in-Progress

The proposed flow enables the generation of synthesizable hardware designs from MLIR-based dataflow descriptions by leveraging the capabilities of the MDC toolchain. The approach bridges high-level, operator-centric specifications in MLIR with the dataflow-oriented hardware generation process supported by MDC.

3.1. Flow Overview

An overview of the proposed work is shown in Figure 1. The pipeline begins with an MLIR program expressed using a combination of standard dialects such as *arith*, *math*, and *scf*, along with the custom

[1]Available on GitHub at https://github.com/mdc-suite/mdc

dfg dialect. This is extensible and linked to what is currently supported by *dfg* and it's High-Level Synthesis (HLS) target. The integration required to slighly modify MLIR interface, by passing as input. A dedicated pass pipeline, *dfg-to-mdc*, integrated within the MLIR infrastructure, transforms these intermediate representations into a dataflow specification compatible with the MDC toolchain—specifically, generating the corresponding *.XDF* and *.CAL*, as described in the section 2.2. In addition to this, the integration flow leverages the existing *emitHLS-generate-project* pass to produce synthesizable Verilog actor modules using Vitis HLS. It relies on an intermediate dialect that maps each construct directly (1:1) to standard C++ expressions or classes, thereby facilitating compatibility with the AMD design toolchains such as Vivado and Vitis HLS. Although no formal publication describing this pass is currently available, it represents a MYRTUS extension of earlier developments, and relevant citations will be included once publicly released [5].

Finally, the resulting hardware was packaged into a Vivado project for simulation and synthesis, enabling verification of functionality and hardware compatibility. The modular nature of the pipeline allows for future enhancements, such as tighter integration with Vitis HLS dialects or user-defined hardware mappings.

3.2. Input Example in MLIR

To demonstrate the capabilities of the proposed compilation flow, we utilize a simple yet representative dataflow application defined in MLIR using the *dfg-mlir* dialect. As shown in Figure 2, the example consists of two computational operators: an incrementer and a left shifter. These are expressed as modular operators using *dfg.operator* and connected via a *dfg.region* to form a small synchronous dataflow (SDF) network. This structure reflects the data-parallel and stateless semantics typical of many dataflow programs.

```
dfg.operator @inc inputs(%in: i16) outputs(%out: i16) {
    %0 = arith.constant 1 : i16
    %1 = arith.addi %in, %0 : i16
    dfg.output %1: i16
}

dfg.operator @lshifter inputs(%in: i16) outputs(%out: i16) {
    %0 = arith.constant 2 : i16
    %1 = arith.muli %in, %0 : i16
    dfg.output %1: i16
}

dfg.region @top inputs(%arg0: i16) outputs(%arg1: i16) {
    %0:2 = dfg.channel(1) : i16
        dfg.instantiate @inc inputs(%arg0) outputs(%0#0) : (i16) -> i16
        dfg.instantiate @lshifter inputs(%0#1) outputs(%arg1) : (i16) -> i16
}
```

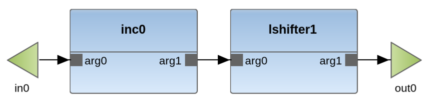
Figure 2: A MLIR program example

Each operator defines a single input and output, and performs an elementary arithmetic operation using standard *arith* dialect instructions. The incrementer adds a constant value to the input, while the shifter shifts it one bit to the left. These operations are defined independently and later instantiated in the *@top*[2] region to form a complete graph.

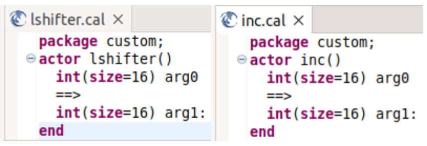
The conversion is performed using *dfg-to-mdc* pass pipeline integrated into the MLIR toolchain, as illustrated in Figure 1. This pipeline processes programs written in the *dfg-mlir* dialect—such as the example shown in Figure 2—and automatically generates the project structure required by the MDC toolchain. Specifically, it produces a *.XDF* file that describes the top-level dataflow graph corresponding to the *top* region, along with two *.CAL* files representing the *inc* and *lshifter* operators. This process is fully automated and requires no manual intervention, allowing a seamless transition from MLIR-level modeling to MDC-level dataflow specifications. Figure 3 illustrates a representative generated network composed of two MLIR-defined actors: an increment operator *(inc0)* followed by a left shifter *(lshifter1)*. As shown, the structural composition and actor granularity from MLIR are preserved in the generated MDC view, reinforcing traceability and modularity.

In the next step, the *emitHLS* dialect is employed for lowering the input MLIR program to evetually synthesize Verilog actor modules from their corresponding MLIR definitions. To better support downstream integration with the MDC flow, the pass has been extended with a configuration flag *–for-MDC*. When this flag is enabled, the pass generates only the essential HLS-related artifacts—namely, the C++ source files and associated TCL scripts—for individual operators, rather than producing a complete

[2]MLIR guarantees identifiers never collide with keywords by prefixing identifiers with a sigil (e.g. %, #, @, ^, !).



(a) XDF file for network topology representation.

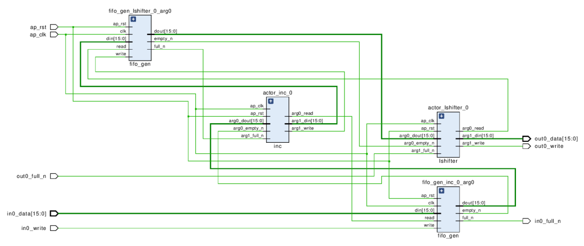


```
package custom;
actor lshifter()
  int(size=16) arg0
  ==>
  int(size=16) arg1:
end
```

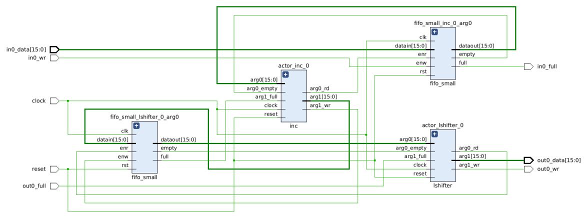
```
package custom;
actor inc()
  int(size=16) arg0
  ==>
  int(size=16) arg1:
end
```

(b) CAL files for actors interfaces.

Figure 3: Representation of the generated files for MDC.



(a) Automatic method



(b) Manual method

Figure 4: Resulting processing blocks after synthesis

system-level design. Using this configuration, synthesizable Verilog files are generated for the *inc* and *lshifter* operators by Vitis HLS. Once these components are generated, the complete hardware description can be assembled by the MDC Core. This modular infrastructure promotes design reuse and extensibility, thereby establishing a robust foundation for scalable and customizable hardware accelerator generation.

3.3. Hardware Generation and Integration in Vivado

Once the MLIR program is translated into MDC-compatible dataflow models, the MDC toolchain is used to generate synthesizable Verilog code. Each operator, such as *@inc* and *@lshifter*, is compiled into a dedicated hardware module, and FIFO-based buffering is automatically inserted by the Platform Composer between operators to preserve dataflow semantics. In addition to these modules, MDC automatically provides the top-level module, FIFO implementations, and a testbench based on the input dataflow specification. These generated files, along with the Verilog actor modules, can be imported directly into a Vivado project for synthesis and simulation without any manual modification, apart from providing a basic constraints file.

Figure 4a illustrates the synthesized Verilog blocks from the automated flow, which follows the Vitis HLS communication protocol (`ap_clk`, `ap_rst`, `_empty_n`, `_full_n`, `_read`, `_write`) and employs a FIFO architecture generated by the *EmitHLS* pass. For comparison, the manual implementation (Figure 4b) adopts the default MDC protocol from the reference repository, using a similar handshake (`_wr`, `_rd`, `_full`, `_empty`) with custom FIFOs. While the signal names and active-high/low conventions differ, the two protocols are functionally equivalent, ensuring that both designs can be fairly compared.

Table 1
Post-Synthesis Resource Utilization on Kria KV260

Method	LUTs	FFs	LUTRAMs	Execution time(cycle)	fmax(MHz)
Automated	115 (0.10%)	63 (0.03%)	38 (0.07%)	65	190
Manual	116 (0.10%)	63 (0.03%)	38 (0.07%)	65	165

To quantitatively assess the two approaches, post-synthesis resource utilization results targeting the Kria KV260 platform are summarized in Table 1. As shown, both the automated pipeline and the manually written HDL consume nearly identical hardware resources, with only negligible differences—likely due to backend tool optimizations. Notably, both implementations achieve the same execution times for 64-samples inputs, while the automated version achieves a slightly higher maximum clock frequency (190MHz vs. 165 MHz), suggesting improved timing closure. These results confirm that the MDC toolchain reliably produces synthesizable, functionally equivalent Verilog designs in both manual and automated modes, seamlessly integrating with Vivado for simulation and synthesis. To quantitatively assess the two approaches, post-synthesis resource utilization results targeting the Kria KV260 platform are summarized in Table 1. As shown, both the automated pipeline and the manually written HDL consume nearly identical hardware resources, with only negligible differences—likely due to backend tool optimizations. Notably, both implementations achieve the same execution times for 64-samples inputs, while the automated version achieves a slightly higher maximum clock frequency (190MHz vs. 165 MHz), suggesting improved timing closure. These results confirm that the MDC toolchain reliably produces synthesizable, functionally equivalent Verilog designs in both manual and automated modes, seamlessly integrating with Vivado for simulation and synthesis.

3.4. Discussion and Preliminary Observations

The proposed flow demonstrates the feasibility of generating correct and synthesizable hardware designs directly from MLIR-based dataflow specifications using the MDC toolchain. Both manual and automated approaches lead to functionally equivalent hardware modules, confirming the correctness of the transformation pipeline and the reliability of MDC's backend infrastructure. The manual method offers greater control and flexibility, but requires additional effort in preparing the input files. In contrast, the automated flow—enabled by the proposed *dfg-mlir* pipeline combined to MDC—substantially improves productivity by automatically generating all required files, making it more suitable for scalable and iterative development.

Simulation and synthesis results confirm functional correctness and modularity in both approaches, validating the viability of integrating dataflow-based compilation into MLIR for FPGA accelerator design. Importantly, this work introduces the first integrated flow for CGR datapath/platforms within the MLIR ecosystem. While our evaluation focuses on a simple dataflow kernel, the methodology is inherently extensible: the abstractions in *dfg-mlir* and the backend flexibility of MDC are not limited to small examples and can naturally scale to more complex applications and larger reconfigurable platforms.

4. Conclusion and Future Work

In this work, we presented the integration of the MDC toolchain with the MLIR infrastructure, enabling a unified and extensible compilation path for coarse-grain reconfigurable hardware accelerators. This integration was realized by adding MDC as a target within the *dfg-mlir* project, allowing MDC to be targeted from MLIR-based compiler pipelines. To validate the integration, we evaluated a simple dataflow kernel and observed no performance degradation when comparing the automatically generated implementation with its manually optimized counterpart. This demonstrates the feasibility and effectiveness of the compilation flow targeting MDC, without compromising the performance benefits typically associated with hand-crafted designs.

Future work will focus on extending the evaluation to a broader and more diverse set of dataflow applications, ranging from signal processing kernels to machine learning operators, with the aim of generating reconfigurable accelerators that fully exploit MDC's resource-sharing capabilities. This will allow us to assess scalability not only in terms of synthesis feasibility but also in terms of throughput, area efficiency, and design productivity for larger, more realistic workloads. In parallel, we plan to embed the resource-sharing algorithm directly into *dfg-mlir* as a dedicated optimization pass, making it natively accessible to other MLIR-based compilation flows. This integration will enrich the ecosystem with coarse-grain reconfigurability as a first-class optimization option, enabling systematic exploration of trade-offs between performance, area, and reconfigurability. Crucially, these directions demonstrate that scaling to large designs is technically feasible and not constrained by the underlying tools, but rather offers new opportunities for co-design across compiler and hardware layers.

Acknowledgments

This work was supported by MYRTUS. "MYRTUS is funded by the European Union, by grant No. 101135183. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them."

Declaration on Generative AI

During the preparation of this work, the authors used ChatGPT in order to: improve grammar, spelling and general readability. After using these tool(s)/service(s), the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] F. Palumbo, F. Ratto, C. Rubattu, M. K. Zedda, T. Fanni, V. Rao, B. Driessen, J. Castrillon, Multi-partner project: Key enabling technologies for cognitive computing continuum-myrtus project perspective, in: 2025 Design, Automation & Test in Europe Conference (DATE), IEEE, 2025, pp. 1–7.

[2] C. Sau, T. Fanni, C. Rubattu, L. Raffo, F. Palumbo, The multi-dataflow composer (mdc) tool suite, https://mdc-suite.github.io/mdc, 2021. Open-source framework for generating coarse-grain reconfigurable (CGR) hardware architectures.

[3] O. D. Team, Orcc: Open rvc-cal compiler workshop and code release, https://github.com/orcc/orcc, 2025. Open-source compiler for RVC-CAL actor specifications and dataflow graph generation.

[4] F. Manca, F. Ratto, F. Palumbo, Onnx-to-hardware design flow for adaptive neural-network inference on fpgas, in: International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, Springer, 2024, pp. 85–96.

[5] F. et al., dfg-mlir: An mlir dialect for modelling kpn-style dataflow graphs, https://github.com/Feliix42/dfg-mlir, 2025. Open-source MLIR dialect for dataflow modelling in MYRTUS/EVERSTEP projects.

[6] J. Bi, G. Korol, J. Castrillon, Leveraging the mlir infrastructure for the computing continuum (2024).

[7] C. Lattner, M. Amini, U. Bondhugula, A. Pienaar, R. Riddle, T. Shpeisman, N. V. Cohen, M. Zadok, Mlir: Scaling compiler infrastructure for domain specific computation, 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (2021) 2–14.

[8] E. Consortium, EVEREST D2.3: Intermediate representations and toolchain for hardware generation, Technical Report D2.3, H2020 EVEREST Project, 2021. Available: https://everest-h2020.eu/.

[9] F. Palumbo, M. K. Zedda, T. Fanni, A. Bagnato, L. Castello, J. Castrillon, R. D. Ponte, Y. Deng, B. Driessen, M. Fadda, T. H. du Fretay, J. de Oliveira Filho, V. Rao, F. Regazzoni, A. Rodriguez, M. Schranz, G. Sedda, Myrtus: Multi-layer 360° dynamic orchestration and interoperable design environment for compute-continuum systems, in: Proceedings of the 21st ACM International Conference on Computing Frontiers: Workshops and Special Sessions, CF '24 Companion, Association for Computing Machinery, New York, NY, USA, 2024, p. 101–106.

[10] C. Sau, T. Fanni, C. Rubattu, L. Raffo, F. Palumbo, The multi-dataflow composer tool: An open-source tool suite for optimized coarse-grain reconfigurable hardware accelerators and platform design, Microprocessors and Microsystems 80 (2021) 103326.

[11] C. C. d. Souza, A. M. Lima, G. Araujo, N. B. Moreano, The datapath merging problem in reconfigurable systems: Complexity, dual bounds and heuristic evaluation, Journal of Experimental Algorithmics 10 (2005) 2.2–es.

[12] Ieee standard for systemverilog–unified hardware design, specification, and verification language, IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012) (2018) 1–1315.