Deterministic Allocation and Scheduling for Buffered Exposed Datapath Architectures

Nadine Kercher[0009−0007−0300−0689] and Klaus Schneider[0000−0002−1305−7132]

Department of Computer Science,
RPTU University Kaiserslautern-Landau, Germany
https://es.cs.rptu.de

Abstract. Exposed datapath architectures reveal their internal architecture, enabling the compiler to allocate processing units (PUs) for the operations of a dataflow graph, and to schedule them together with the data transfers of their intermediate results between the PUs. Following traditional compilers, we split the allocation and scheduling processes into separate phases, although optimal solutions are lost this way. In this paper, we assume that a schedule must be determined for a given allocation. We prove that if all nodes mapped to a PU are completely ordered, a schedule can be inferred efficiently by constraint propagation alone. Allocations that satisfy this property are called deterministic allocations and are the basis for highly efficient code generation. In addition to the theory of deterministic allocations, we present three significant classes of deterministic allocations with their advantages and disadvantages.

1 Introduction

Exposed datapath architectures such as RAW/Tilera, TRIPS, DySER, Tartan, Conservation Cores, Transport-Triggered Architectures, STA, FlexCore, MOVE-Pro, AMIDAR, and SCAD expose their internal architecture to the compiler which can then exploit instruction-level parallelism (ILP) by (1) allocating processing units (PUs) for the operations of a dataflow graph, (2) scheduling of execution on the PUs, and (3) scheduling the communication of intermediate results between PUs. The paradigm of exposed datapath architectures reduces the processor to a network of simple PUs so that the processor's circuit size scales linearly with the number of PUs. *Buffered exposed datapath* (BED) architectures replace global register files with FIFO buffers at the network ports of the PUs. Therefore, the execution of basic blocks on BED architectures can follow data dependencies to fully exploit ILP.

BED processors can easily be implemented with a large number of PUs [2,4]. However, generating code for BED processors is more difficult because it requires PU allocation, instruction scheduling, and communication of intermediate results between PUs. In particular, the compiler must ensure that values in buffers are accessed in FIFO order which can be violated if different nodes of the dataflow graph are mapped to the same PUs. In [7,6], SAT/SMT constraints were formulated to map dataflow graphs onto BED architectures for optimal compilation with SAT/SMT solvers.

Since the overall problem is NP-complete, as is register allocation for RISC processors, heuristics are needed to solve it efficiently. Following traditional compilers,

which solve register allocation and instruction scheduling separately, [1] suggested to split the PU allocation and scheduling problems for BED architectures also into separate phases. Several instances of PU allocation can be solved in polynomial time, and the scheduling problem for a given PU allocation reduces to SAT constraints, most of which are 2-SAT clauses (which can also be solved in linear time). However, like the general SAT solving procedure, also the scheduling procedure has to make guesses with backtracking in unsuccessful cases.

In this paper, we consider the scheduling problem for a given PU allocation. Our main result, Theorem 1, shows that PU allocations that generate total orders per PU produce SAT constraints that allow our constraint solver to derive a schedule *without backtracking*. These PU allocations are called deterministic and are essential for an efficient code generator. To demonstrate the practical usefulness of the theorem, we present three deterministic PU allocations.

The paper is organized as follows: The next section provides a brief review of BED architectures and their SAT constraints for PU allocation and instruction scheduling. Section 3.1 proves the paper's main contribution: deterministic allocations, which map to each PU a totally ordered set of nodes, allow the construction of a schedule without backtracking. Sections 3.2, 3.3, and 3.4 present three practically useful deterministic PU allocations. Section 4 presents experimental results for the allocations presented. Finally, Section 5 summarizes the main contributions.

2 Preliminaries

A general template for a *buffered exposed datapath (BED)* architecture is shown in Fig. 1: A BED processor consists of a (large) number of PUs, a load/store unit (LSU) for accessing data memory, and a control unit (CU) for accessing program memory. The PUs, LSU, and CU are connected via FIFO buffers through a network on-chip to avoid unnecessary PU synchronization. The decentralization of all components of BED architectures results in a linear circuit size in terms of the PU number. For more information, see [1,8,7,6].

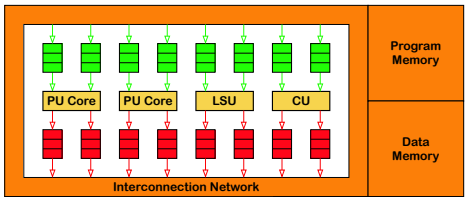


Fig. 1: A general BED architecture.

Some BED processors, such as SCAD and TTA, execute move instructions that transfer a constant (an immediate operand or an opcode) or a result value from an output buffer to an input buffer. Due to the production/consumption of data tokens that BED processors share with dataflow computing, the code generation typically uses dataflow graphs (DFGs) as an intermediate compiler representation [5,7]. Generating move code from DFGs requires (1) a mapping of the DFG nodes to PUs (PU allocation), (2) an ordering of the DFG nodes for instruction scheduling, and (3) an ordering of the DFG edges for a communication schedule [1,7,6]. For a given PU allocation, the SAT constraints for specifying (2) and (3) were specialized in [1] as follows:

Definition 1 (SAT Constraints for a Given Allocation). *For an arbitrary dataflow graph with nodes $\mathcal{P}$ and buffers $\mathcal{B}$, and a given allocation $\alpha : \mathcal{P} \to \{1, \ldots, \varrho\}$ mapping nodes $\mathcal{P}$ to PUs $P_1, \ldots, P_\varrho$, we define the following constraints using a strict ordering relation $\prec$ on the nodes, and a strict ordering relation $\sqsubset$ on the buffers:*

- *data dependency constraints: for every edge $b : p \to q$, we demand $p \prec q$*
- *FIFO behavior constraints: for all edges $b_i : p_i \to q_i$ and $b_j : p_j \to q_j$, we demand*
  - *$p_i \prec p_j \leftrightarrow b_i \sqsubset b_j$ if b_i and b_j use the same output buffer of the same PU*
  - *$q_j \prec q_i \leftrightarrow b_j \sqsubset b_i$ if b_i and b_j use the same input buffer of the same PU*

Data dependency constraints ensure that nodes are scheduled according to their data dependencies. FIFO behavior constraints guarantee that the order in which values are written to and read from a FIFO buffer is the same. Solutions to these constraints correspond with instruction and data transfer schedules.

The constraint solver presented in [1] generates an undirected graph whose vertices are labeled with node or buffer order constraints. If the equivalence $p_i \prec p_j \leftrightarrow b_i \sqsubset b_j$ is derived from the FIFO constraints by transitive closure, then the vertices $p_i \prec p_j$ and $b_i \sqsubset b_j$ are connected. Therefore, vertices in the same strongly connected component must have the same truth value, which is determined by either data dependency constraints or *guesses of the solver*. As shown in [1], *these guesses may be unsuccessful even if a solution exists, so that backtracking to previous guesses is generally unavoidable.*

3 Scheduling with Deterministic Allocations

3.1 Deterministic Allocations

A PU allocation is called *deterministic* if the DFG nodes mapped to any PU are *totally ordered*. For such deterministic allocations, all equivalences can be resolved by constraint propagation alone, i.e., no guessing is required:

Theorem 1 (Scheduling with Deterministic Allocations). *For any deterministic PU allocation (that maps all nodes of a dataflow graph to PUs such that the set of nodes mapped to the same PU is totally ordered), the constraint solver can propagate the facts by transitive closure computation without guessing and backtracking.*

Proof. The solver must find partial orders for the nodes and buffers that satisfy the constraints given in Definition 1. Note that the data dependencies are independent of the allocation, but the FIFO behavior constraints depend on it. According to the FIFO behavior constraints, equivalences are created for a pair of buffers if (1) their source corresponds with the same output buffer of a PU or (2) their target corresponds with the same input buffer of a PU. Thus, for each equivalence $p_i \prec p_j \leftrightarrow b_i \sqsubset b_j$, the truth value of $p_i \prec p_j$, and consequently also the truth value of $b_i \sqsubset b_j$, is determined by the given total node order. After processing all equivalences in this manner, the transitive hull of the result is checked. If irreflexivity and transitivity are satisfied, the result is valid; otherwise, a conflict is found. In case of a conflict, there is no valid solution since no guessing was done. □

The advantage of deterministic allocations is obvious: there is no need to guess, which would otherwise require backtracking in case of a conflict until a valid solution or proof of unsatisfiability is found. This improves the runtime of the solver to find a schedule for a given allocation. However, this benefit comes at the cost of providing enough PUs such that all pairs of nodes p and q with neither $p \prec q$ nor $q \prec p$ can be assigned to different PUs.

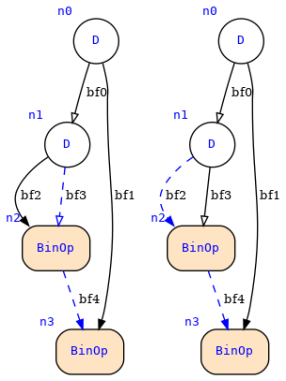
The remainder of this section presents three types of deterministic allocations and their respective advantages and disadvantages. They use different strategies to define a total order for the nodes assigned to a PU ensuring that the solver does not have to guess.

3.2 Vertex-Disjoint Path Cover Allocation

A vertex-disjoint path cover consists of the minimum number of paths needed to cover each vertex of a graph exactly once. A graph can have multiple vertex-disjoint path covers [3]. A PU allocation can now map each of these paths to the same PU. This reduces the necessary communication between the PUs since one of the next input values is always produced by a node mapped to the same PU.

Corollary 1. *For any allocation that associates the paths of a vertex-disjoint path cover of the dataflow graph with the PUs of a BED architecture, either a solution or a conflict can be deduced without guessing and backtracking.*

Figure 2a shows an example graph that leads to a conflict if mapped to a single PU. Nodes labeled with D duplicate their input to the two outputs, and BinOp can be any operation with two inputs and one output. The edges are interpreted as follows: a black arrow encodes a left input of a PU, a blue dashed arrow a right input of a PU, a filled arrowhead a left output of a PU, and an empty arrowhead a right output of a PU.



(a) (b)

Fig. 2: DFG (a) leads to a conflict if all nodes are mapped to the same PU, while DFG (b) solves the problem.

Both edges bf1 and bf2 describe a data transfer from the left output to the left input of the PU. Sharing the same source and target generates the FIFO behavior constraints $\texttt{bf1} \sqsubset \texttt{bf2} \leftrightarrow \texttt{n0} \prec \texttt{n1}$ and $\texttt{bf1} \sqsubset \texttt{bf2} \leftrightarrow \texttt{n3} \prec \texttt{n2}$.

According to the path, the nodes are ordered as follows: $\texttt{n0} \prec \texttt{n1} \prec \texttt{n2} \prec \texttt{n3}$. The order of the producer nodes $\texttt{n0} \prec \texttt{n1}$ results in $\texttt{bf1} \sqsubset \texttt{bf2}$, while the order of the consumer nodes $\texttt{n2} \prec \texttt{n3}$ results in $\texttt{bf2} \sqsubset \texttt{bf1}$. In other words, the production and consumption orders of the two edges that share the same source and target do not match. This creates a conflict in the FIFO buffer which is unavoidable for the given DFG and allocation since only deduction was used without guessing.

Analyzing this more closely, we see that the problem stems from the transitive edge `bf1` which shares the same source and target as the aforementioned path. This can be solved by modifying the graph as shown in Figure 2b where the outputs of one of the duplication nodes are swapped so that all edges have different sources or targets than the transitive edge.

Now, the valid buffer order $\texttt{bf0} \sqsubset \texttt{bf3} \sqsubset \texttt{bf1} \sqsubset \texttt{bf2} \sqsubset \texttt{bf4}$ can be derived. However, not all cases are solvable. Since there are only four possibilities to connect different sources with different targets, having four transitive edges (five parallel edges) always results in two edges sharing one virtual channel.

As a result, the vertex-disjoint path cover allocation enables scheduling in polynomial time because it eliminates the need for guessing. Another advantage is that the PUs require less communication since connected paths are mapped to the same PU such that intermediate results can be directly used there. Furthermore, some, but not all, of the conflict cases are solvable. However, if there are fewer PUs than paths, some paths must share a PU. Thus, with a limited number of PUs, it is not always possible to obtain a total order of the nodes mapped to the same PU.

3.3 Per Level Allocation

In leveled graphs, all nodes are assigned to a level such that edges only connect nodes of levels i and $i+1$. Any non-leveled graph can be transformed into a leveled graph by inserting copy nodes with one input and one output. Depending on where the copy nodes are inserted, several leveled versions of a non-leveled graph can be obtained. Special SAT constraints can be defined for leveled dataflow graphs [6], which simplifies their allocation and scheduling. We can also define a simple deterministic allocation for leveled graphs, as explained in the following corollary.

Corollary 2. *For any given allocation that maps at most one node per level of the dataflow graph to the same PU, ordering the nodes of one PU by their levels either results in a conflict-free schedule without guessing or proves that no schedule exists.*

The per-level allocation is even complete, i.e., it allows the solver to *always find a solution if enough PUs are provided:*

Corollary 3. *Consider an allocation that maps at most one node from each level of a dataflow graph to the same PU. If the nodes of a PU are ordered according to their levels, a solution can always be deduced for that allocation without guessing.*

Proof. By Corollary 2, it remains to prove that no conflict can occur. A conflict occurs if the inferred order is not a partial order, i.e., if irreflexivity or transitivity is violated. The order is deduced by propagating the facts step by step through the equivalences. Each equivalence is of the form $p_i \prec p_j \leftrightarrow b_i \sqsubset b_j$ where p_i and p_j are mapped to the same PU, and b_i and b_j use the same input or output buffer of that PU. According to the per-level allocation, p_i and p_j are on different levels, and thus b_i and b_j are between different pairs of levels $(k, k+1)$ and $(l, l+1)$. Now, consider the buffer layers between the levels. These layers are totally ordered by the given total order of the nodes according to their levels. Furthermore, the partial order is not violated by buffers from the same layer, since they are independent of each other. □

In summary, the per-level allocation has the advantage that it does not require guessing and always finds a solution if enough PUs are provided. However, this advantage comes at the cost of leveling the graph, which takes time and makes the dataflow graph (at most quadratically) larger. Unlike the vertex-disjoint path cover allocation, it does not reduce communication between PUs.

3.4 Root Rank Allocation

Finally, we construct a version of the per-level allocation for non-leveled graphs:

Definition 2 (Root Ranks). *The root ranks of the nodes of a directed graph G are determined iteratively: Roots, i.e., nodes without predecessors have rank 0, and for all other nodes, the rank is the maximum rank of the predecssors plus 1.*

Lemma 1. *Nodes in a dataflow graph can only depend on nodes with smaller root ranks.*

Proof. Suppose node n of root rank i depends on node m of root rank $i+k$ with $k > 0$. This means that an output of node m is processed to determine an input of node n. Thus, there exists a path from node m to node n. As long as node m is in the graph, node n cannot be a root, since node m is one of its predecessors. Thus, node m has a lower root rank than node n, which is a contradiction of the assumption. □

For leveled graphs, the root ranks are the levels. However, they are also applicable to non-leveled graphs and do not require edges to connect only root ranks i with root ranks $i + 1$.

Corollary 4. *Assume that a given allocation maps at most one node per root rank of the dataflow graph to different PUs. Ordering a PU's nodes according to their root rank results in constraints for which either a solution or a conflict can be deduced, i.e., no guessing is required for scheduling.*

The root rank allocation can be viewed as a combination of the per-level and the vertex-disjoint path cover allocations. Like the vertex-disjoint path cover allocation, it can be applied to non-leveled graphs and does not require guessing, and it also suffers from transitive edges. It also has the disadvantage of the per-level allocation: it cannot take advantage of efficient bypasses of values from the output of a PU to its input, since the communication between the PUs is not reduced by this allocation.

4 Experimental Results

In our experiments, we use the specialized constraint solver presented in [1]. It is executed on the constraints for the three allocations presented in the previous section. The measured time for solving the resulting scheduling constraints does neither include the translation of programs into dataflow graphs nor the generation of constraints. Our test data files are available on the Averest website[1] and are listed in Table 1. All

[1] http://www.averest.org/

	Vertex-Disjoint Path Cover			Per-Level			Root Rank		
File	PUs	constr.	time [ms]	PUs	constr.	time [ms]	PUs	constr.	time [ms]
BinaryTree_Scl_16	8	30	1.91	8	38	1.8	8	38	1.92
BinaryTree_Scl_32	16	74	6.72	16	97	6.98	16	97	6.86
BinaryTree_Scl_4	2	2	0.01	2	2	0.01	2	2	0.01
BinaryTree_Scl_64	32	166	32.26	32	218	30.67	32	218	30.62
BinaryTree_Scl_8	4	10	0.13	4	12	0.13	4	12	0.13
EvalPolynomial_SclGlb_16	17	815	1052.53	15	5396	25555.12	7	3060	8563.34
EvalPolynomial_SclGlb_32	33	3071	16727.03	18	22276	460052.6	10	11103	105914.32
EvalPolynomial_SclGlb_4	5	130	25.93	5	359	108.6	4	244	47.26
EvalPolynomial_SclGlb_8	9	378	217.68	10	1364	1625.67	6	888	668.18
EvalPolynomial_SclLoc_16	17	815	1050.6	15	5396	25559.71	7	3060	8552.02
EvalPolynomial_SclLoc_32	33	3071	16739.09	18	22276	460825.52	10	11103	106092.84
EvalPolynomial_SclLoc_4	5	130	25.73	5	359	108.87	4	244	47.37
EvalPolynomial_SclLoc_8	9	378	217.38	10	1364	1638.81	6	888	666.43
FastFourierTransform_SclGlb_4	4	58	4.24	5	94	6.66	4	90	5.72
FastFourierTransform_SclGlb_8	9	340	172	11	647	313.32	9	578	258.81
FastFourierTransform_SclLoc_4	4	58	4.47	5	98	6.66	4	94	5.68
FastFourierTransform_SclLoc_8	9	340	168.93	11	645	314.38	9	569	255.14
MatrixMultCannon_SclGlb_2	9	151	29.44	10	353	85.69	8	283	55.08
MatrixMultCannon_SclGlb_3	31	493	268.1	28	1660	2083.87	20	1237	1083.17
MatrixMultCannon_SclGlb_4	73	1632	3174.56	60	4766	17946.54	42	3437	9535.45
MatrixMultCannon_SclLoc_2	12	91	7.69	13	300	60.47	8	205	28.6
MatrixMultCannon_SclLoc_3	36	342	92.55	32	1463	1598.72	21	1021	790.14
MatrixMultCannon_SclLoc_4	80	1214	1547.95	63	4390	14427.34	42	3031	7334.25
MatrixMultSimple_SclGlb_2	8	40	1.15	8	53	1.27	8	53	1.22
MatrixMultSimple_SclGlb_3	27	253	53.71	33	511	143.05	25	427	97.7
MatrixMultSimple_SclGlb_4	64	875	783.9	78	1605	1583.18	55	1428	1298.08
MatrixMultSimple_SclLoc_2	8	40	1.16	8	52	1.08	8	52	1.08
MatrixMultSimple_SclLoc_3	27	252	54.96	33	509	145.59	25	425	100.53
MatrixMultSimple_SclLoc_4	64	885	766.15	78	1608	1533.7	55	1424	1234.7
MatrixMultStrassenWinograd_SclGlb_4	63	3831	21514.08	78	19491	337556.72	38	9938	83643.25
ParallelPrefixTree_SclGlb_16	9	298	139.44	15	2364	4626.78	8	727	443.56
ParallelPrefixTree_SclGlb_32	17	714	791.98	31	9808	80198.79	16	2245	4257.38
ParallelPrefixTree_SclGlb_4	2	15	0.4	3	34	0.85	2	23	0.41
ParallelPrefixTree_SclGlb_64	33	1682	4400.1	63	34270	1100875.25	32	5817	29176.61
ParallelPrefixTree_SclGlb_8	5	90	12.97	7	426	146.22	4	175	26.23
ParallelPrefixTree_SclLoc_16	8	302	146.29	15	1833	2904.45	8	699	414.4
ParallelPrefixTree_SclLoc_32	16	764	907.44	31	6779	38037.68	16	1988	3408.16
ParallelPrefixTree_SclLoc_4	2	22	0.85	3	52	2.4	2	33	1.03
ParallelPrefixTree_SclLoc_64	32	1758	5197.79	63	22001	432511.45	32	4863	20538.76
ParallelPrefixTree_SclLoc_8	4	100	15.67	7	393	126.14	4	197	33.01
TransHull_SclGlb_2	5	301	142.13	6	1607	2234.34	3	813	572.88
TransHull_SclGlb_3	16	1639	4364.46	14	14024	168883.45	6	5160	22188.99
TransHull_SclGlb_4	37	3466	19150.24	26	52235	2771997.32	11	18739	304298.32
TransHull_SclLoc_2	5	243	90.19	6	1463	1870.58	3	697	428.07
TransHull_SclLoc_3	14	1363	2897.69	14	12350	131432.83	6	4858	20079.19
TransHull_SclLoc_4	33	4391	31714.01	25	55313	3107344.56	12	18697	303291.22

Table 1: Number of PUs, the number of constraints and the average runtime for the test files with the three different allocations presented in the previous section.

experiments were performed on a 64-bit machine running the Ubuntu 22.04 operating system with 192GB RAM and two sockets, each containing an Intel Xeon Gold 5220R CPU.

The experimental results are given in Table 1, which lists the number of required PUs, the number of constraints to be solved and the average solver runtime for each of the presented allocations on different test files. For one DFG, the different allocations result in different numbers of constraints. The slowest runtime for the per-level allocation is about 3100 seconds (55313 constraints), while the root rank allocation takes at most about 304 seconds (18739 constraints) and the vertex-disjoint path cover allocation takes only about 32 seconds (4391 constraints).

In addition to runtime, practicality also depends on the number of PUs required and whether the resulting constraints are satisfiable. The root rank allocation requires the fewest PUs. Compared to the per level allocation, it is the same for leveled graphs and can be improved if considering non-leveled graphs. Compared to the vertex-disjoint path cover allocation, two nodes of the same root rank cannot be on the same path.

The per level and the vertex-disjoint path cover allocation do not infer bounds on each other. Regarding satisfiability, the per level allocation is best because it is always satisfiable. For our test files, the vertex-disjoint path cover allocation returned 'sat' for 40 while the root rank allocation only returned 'sat' for 12 out of 46 files. According to our experiments, the vertex-disjoint path cover allocation is the best in terms of practicality.

5 Conclusions

This paper considers the SAT constraints for mapping a dataflow graph with a given allocation to a number of FIFO buffered PUs. For the nodes and for the buffers, a strict and total order must be found that satisfies the data dependencies and the FIFO behavior constraints. Except for transitivity, these constraints are 2-SAT clauses and can be solved efficiently by the specialized constraint solver presented in [1]. However, this constraint solver may require guessing and backtracking for arbitrary PU allocations.

In this paper, deterministic PU allocations are considered, i.e., PU allocations where all nodes mapped to the same PU are totally ordered. We have shown that for deterministic PU allocations, it is possible to construct the required schedule in polynomial time by constraint propagation alone, i.e., without guessing and backtracking as usual in constraint solving.

References

1. Bhagyanath, A., Kercher, N., Schneider, K.: Allocation and scheduling of dataflow graphs on hybrid dataflow/von Neumann architectures. In: Brandt, J., Zhu, Q. (eds.) Formal Methods and Models for Codesign (MEMOCODE). IEEE Computer Society, Hamburg, Germany (2023)
2. Burger, D., Keckler, S., McKinley, K., Dahlin, M., John, L., Lin, C., Moore, C., Burrill, J., McDonald, R., Yoder, W.: Scaling to the end of silicon with EDGE architectures. IEEE Computer **37**(7), 44–55 (July 2004)
3. Kercher, N.: Code Generation for Buffered Exposed Datapath Architectures. Master's thesis, Department of Computer Science, RPTU Kaiserslautern-Landau, Kaiserslautern, Germany (July 2023)
4. Sankaralingam, K., Nagarajan, R., Liu, H., Kim, C., Huh, J., Ranganathan, N., Burger, D., Keckler, S., Mcdonald, R., Moore, C.: TRIPS: A polymorphous architecture for exploiting ILP, TLP, and DLP. ACM Transactions on Architecture and Code Optimization **1**(1), 62–93 (2004)
5. Schneider, K.: Translating structured sequential programs to dataflow graphs. In: Formal Methods and Models for Codesign (MEMOCODE). pp. 66–77. ACM, Beijing, China (2021)
6. Schneider, K., Bhagyanath, A.: Consistency constraints for mapping dataflow graphs to hybrid dataflow/von Neumann architectures. Transactions on Embedded Computing Systems (TECS) **22**(5), 81:1–81:25 (2023)
7. Schneider, K., Bhagyanath, A., Roob, J.: Code generation criteria for buffered exposed datapath architectures from dataflow graphs. In: Languages, Compilers, and Tools for Embedded Systems (LCTES). pp. 133–145. ACM, San Diego, CA, USA (2022)
8. Schneider, K., Bhagyanath, A., Roob, J.: Virtual buffers for exposed datapath architectures. In: Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV). ITG-Fachbericht, vol. 302, pp. 45–55. VDE (2022)