
# Code Generation Criteria for Buffered Exposed Datapath Architectures from Dataflow Graphs

Klaus Schneider
Department of Computer Science
University of Kaiserslautern
Germany
schneider@cs.uni-kl.de

Anoop Bhagyanath
Department of Computer Science
University of Kaiserslautern
Germany
anoop@rhrk.uni-kl.de

Julius Roob
Department of Computer Science
University of Kaiserslautern
Germany
roob@cs.uni-kl.de

## Abstract

Many novel processor architectures expose their processing units (PUs) and internal datapaths to the compiler. To avoid an unnecessary synchronization of PUs, the datapaths are often buffered which results in buffered exposed datapath (BED) architectures. This paper suggests a code generation technique for BED architectures from dataflow graphs that are used as intermediate program representations. Inspired by results on queue layouts in graph drawing, we determine in this paper constraints for the node and edge orderings of the dataflow graphs to ensure the first-in-first-out behavior of the buffers. Formalizing these constraints in propositional logic enables SAT solvers to compute optimal PU allocations. Moreover, future code generation techniques may develop heuristics based on the code generation criteria of this paper.

***CCS Concepts:*** • **Software and its engineering** → **Data flow architectures**; **Compilers**; • **Computer systems organization** → **Parallel architectures**; **Data flow architectures**; **Heterogeneous (hybrid) systems**; *Stack machines*; • **Hardware** → *Hardware accelerators.*

***Keywords:*** dataflow graphs, code generation, exposed datapath architectures, linear graph layouts, queue layouts

**ACM Reference Format:**
Klaus Schneider, Anoop Bhagyanath, and Julius Roob. 2022. Code Generation Criteria for Buffered Exposed Datapath Architectures from Dataflow Graphs. In *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '22), June 14, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3519941.3535076


*LCTES '22, June 14, 2022, San Diego, CA, USA*

ACM ISBN 978-1-4503-9266-2/22/06.
https://doi.org/10.1145/3519941.3535076

## 1 Introduction

Most commercial processors are RISC processors that access the main memory exclusively by load/store instructions while all other instructions use registers for their operands and results. With increasing chip sizes, it became possible to integrate many processing units (PUs) on a single processor core to exploit the instruction-level parallelism (ILP) of programs. However, the so-far used code generators focus on a minimal use of registers instead of an optimal use of ILP. Moreover, the use of ILP is limited by the number of available registers since each operation finally has to store its result in a register. Increasing the number of registers, however, does not scale well with the chip size.

Recent processor architectures like RAW/Tilera [75, 76, 78], TRIPS [12, 66], DySER [39], TTAs [14–16, 48, 52], AMIDAR [38], and SCAD [2, 5–7, 69] are hybrid dataflow architectures [8, 11, 49, 80]: They execute sequential programs with a control-flow (i.e., a single program counter and branch instructions), but use more or less explicitly dataflow graphs aka dataflow process networks (DPNs) [60] as instructions. Some architectures like WaveScalar [72–74] are even pure dataflow machines without a program counter. To avoid unnecessary synchronization between the PUs, FIFO buffers (queues) are often used at the I/O ports of the PUs resulting in buffered exposed datapath (BED) architectures. Avoiding the use of registers at all reveals similarities to classic *queue machines* (see Section 2).

The code generation for BED architectures must rethink current compiler techniques: The prevalent depth-first traversal of syntax trees minimizes the use of registers [33, 70], but limits the use of ILP. To increase the use of ILP, it would be much better to evaluate expression trees level-by-level. However, there is still a big problem for BED architectures: We have to obey the FIFO principle of their buffers which means that we can only access the head elements of the input buffers as operands for executing the next instruction. For expression trees, it is easily seen that a level-by-level left-to-right traversal yields an ordering of the nodes such that all values can be piped through a single queue in such a way that the next operation can access its operands from the head of the queue and can add its result to the tail of the queue. For directed acyclic graphs, however, the problem becomes much more difficult (i.e., NP-complete).

To generalize these algorithms from expression trees to entire programs, it is suggested in this paper to use DPNs for BED code generation. To this end, we consider DPNs with a suitable set of nodes (see Table 1) that are powerful enough to allow a translation from sequential programs to these DPNs [68]. Inspired by results from graph drawing, we then show that the *DPNs should be leveled* which essentially means that a *schedule for the nodes* is determined in such a way that all operands of a node are produced at the same level in the schedule. Leveling can be easily done by adding copy nodes on edges where needed. For the following *allocation of PUs* for the nodes, it is important to analyze *edge crossings in the leveled DPN*. As a main result of this paper, we prove that *edge crossings that use the same 'virtual channel' (see Definition 1) have to be avoided by the node ordering and PU allocation*. For edge crossings that refer to different virtual channels, we generate code sequences that ensure that all operations will find their operands as heads of the input buffers. We use these insights to precisely formulate correct and complete code generation criteria as SAT constraints to enable SAT solvers to derive move code with a minimal number of PUs. Our code generation criteria also provide a basis for code generation heuristics, since any PU assignment and node ordering must avoid these critical edge crossings.

The outline of the paper is as follows: Section 2 briefly explains queue and BED architectures and points out the notion of *virtual channels* that will become important in the later sections. As intermediate program representation for BED architectures, we use leveled DPNs using the nodes listed in Table 1 in Section 3 [68]. In Section 4, we review results from queue layouts in graph drawing. The core of the paper is Section 5 where we prove how code can be generated by avoiding edge crossings in the DPNs whose edges refer to the same virtual channel. Section 6 finally formulates our code generation criteria as SAT constraints to provide a basis for BED code generation.

## 2 Queue and BED Machines

### 2.1 Queue Machines

Queue machines were initially considered as an abstract machine model that is as powerful as Turing machines [9, 77]. A queue machine is essentially a finite state machine with an additional unbounded FIFO queue as external memory. While running, the queue machine may consume values from the head of its queue and may produce new values to be added to the tail of its queue. Pioneering work on the theory of queue machines is found in [9, 13, 77] where it has been shown that they are as powerful as Turing machines. Further work on the theory of queue machines is found in [10, 13, 59] where also applications for real-time computing were considered.

Earliest work on *code generation for queue machines* has been done by [34] where it has already been recognized that code must be generated by a level-wise left-to-right or right-to-left traversal of expression DAGs. Proceeding this way, the operands for each instruction are always found at the head of the queue when needed. They also observed that several of the instructions can be executed in parallel where all required operands are read at once from the queue, then the instructions are all executed in parallel, and finally all the results are concatenated to the tail of the queue (in the right order).

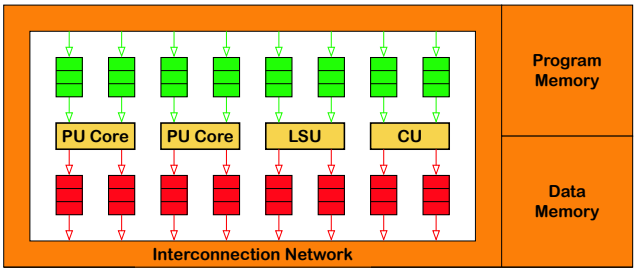


**Figure 1.** General Template of a BED Architecture.

A first hardware implementation (even though only presented as a software model) has been reported in [64, 65] where also the mapping of acyclic dataflow graphs to queue programs is discussed. However, the machine described in [64, 65] is not really a queue machine, since results can also be stored in registers and can be moreover inserted anywhere in the queue. The reason for this was obviously the lack of a code generator for true queue machines as described in [67].

In [67], a hardware synthesis based on dataflow graphs and queue machines is described that is based on results on queue layouts in graph drawing as mentioned in Section 4. In particular, the DPNs were made level-planar with suitable copy and swap nodes to avoid all edge crossings. According to Theorem 8, this allows one to produce queue code by a level-by-level left-to-right traversal.

### 2.2 BED Architectures

Buffered exposed datapath (BED) architectures are generalizations of queue machines for parallel computation. Therefore, BED architectures have many processing units (PUs) whose input and output ports have FIFO buffers as shown in Figure 1. Every buffer has a unique address so that the program of a BED architecture can tell each PU from which output buffers the next values for its input buffers will come from, and also to which input buffers the PU has to send its outputs. Besides the PUs for executing arithmetic-logical instructions, a BED architecture also has a control unit (CU) for fetching instructions from the program memory and issuing these to the PUs, and moreover at least one load/store unit (LSU) to access the main memory. The compiler will determine the data transports from output buffers to input buffers as well as the allocation of PUs for the instructions.

In the most explicit form, the program of a BED architecture consists of a sequence of move instructions src → tgt that denote a data transport from output buffer src to input buffer tgt. Such move instructions were originally introduced for transport-triggered architectures [14–16, 48, 52], but can be used in any BED architecture as well. Using the program counter, the control unit (CU) of the BED architecture will fetch the next move instruction src → tgt from the instruction memory and will make it available to the PUs $\mathcal{P}_{\text{src}}$ owning output buffer src and $\mathcal{P}_{\text{tgt}}$ owning input buffer tgt. $\mathcal{P}_{\text{src}}$ will then send the value at the head of its output buffer src through the interconnection network to input buffer tgt of PU $\mathcal{P}_{\text{tgt}}$ which will add that value to its tail. If output buffer src should be empty (because the value has not yet been computed), $\mathcal{P}_{\text{src}}$ will remember to send that value to address tgt as soon as the value is available, and also $\mathcal{P}_{\text{tgt}}$ will remember to receive that value from address src as the next value of input buffer tgt.

If the required operands of an instruction are available at the heads of the input buffers of the PU executing that instruction, then these *operands are consumed and the output values are produced.* These output values are then concatenated to the tails of the corresponding output buffers. Note that the execution of the PUs and the data transports may be decoupled in time since the PUs can remember the required data transports if the values are not yet available and will perform them as soon as possible.

As a parallel architecture, BED architectures will typically issue many move instructions at any point of time, and hence, also many instructions are executed at any point of time to exploit the ILP of the programs. It is also possible to issue a bundle of move instructions similar to VLIW processors by the CU to the corresponding PUs at once. The interconnection network must therefore be able to connect any output buffer with any input buffer. Preferably, nonblocking networks should be used that allow one to implement any permutation of I/O addresses at any point of time (see [50]).

Since any output buffer src of a PU $\mathcal{P}_{\text{src}}$ can send values to any input buffer tgt of the same or another PU $\mathcal{P}_{\text{tgt}}$, there are many software-controlled configurable datapaths in a BED architecture:

**Definition 1** (Virtual Channels)**.** *A BED architecture with PUs having input buffers $\mathcal{I} = \{I_1, \ldots, I_m\}$ and output buffers $\mathcal{O} = \{O_1, \ldots, O_n\}$ has the virtual channels $\mathcal{I} \times \mathcal{O}$ referring to move instructions $\{O_i \rightarrow I_j \mid O_i \in \mathcal{O}, I_j \in \mathcal{I}\}$.*

In [69], we have shown how to avoid a quadratic growth of the chip size by implementing the above channels using *'virtual buffers'* such that the chip size only grows linearly with the number of PUs.

Finally, note that a PU may implement any kind of function with an arbitrary number of inputs and outputs. In this paper, we consider universal PUs that are capable to execute the operations of the DPN nodes listed in Table 1 with typical arithmetic-logic operations $\odot$. To that end, the PUs require also an input buffer for the instruction's opcode.

**Table 1.** Syntax and Semantics of used DPN Nodes

| syntax | semantics |
|---|---|
| **token control** | |
| $(y) := \mathsf{C}(x)$ | $t = \text{get}(x)$<br>$\text{push}(t, y)$ |
| $(y_0, y_1) := \mathsf{D}(x)$ | $t = \text{get}(x)$<br>$\text{push}(t, y_0)$<br>$\text{push}(t, y_1)$ |
| $(y_0, y_1) := \mathsf{S}(x_0, x_1)$ | $t_0 = \text{get}(x_0)$<br>$t_1 = \text{get}(x_1)$<br>$\text{push}(t_1, y_0)$<br>$\text{push}(t_0, y_1)$ |
| $(y) := \mathsf{J}(x_0, x_1)$ | $t_0 = \text{get}(x_0)$<br>$t_1 = \text{get}(x_1)$<br>$\text{push}(t_0, y)$ |
| $() := \mathsf{K}(x)$ | $t = \text{get}(x)$ |
| **control flow** | |
| $(y) = \mathsf{SEL}(c, x_1, x_0)$ | $t_c = \text{get}(c)$<br>$t_d = \text{if}(t_c)$ then $\text{get}(x_1)$<br>else $\text{get}(x_0)$<br>$\text{push}(t_d, y)$ |
| $(y_1, y_0) = \mathsf{SWT}(c, x)$ | $t_c = \text{get}(c)$<br>$t_x = \text{get}(x)$<br>$\text{if}(t_c)$ then $\text{push}(t_x, y_1)$<br>else $\text{push}(t_x, y_0)$ |
| **memory access** | |
| $(tk_{out}, y) = \mathsf{LD}_a(adr, tk_{in})$ | $t_a = \text{get}(adr)$<br>$t_m = \text{get}(tk_{in})$<br>$\text{push}(\text{mem}[a, t_a], y)$<br>$\text{push}(t_m, tk_{out})$ |
| $(tk_{out}) = \mathsf{ST}_a(adr, tk_{in}, x)$ | $t_a = \text{get}(adr)$<br>$t_m = \text{get}(tk_{in})$<br>$t_x = \text{get}(x)$<br>$\text{mem}[a, t_a] := t_x$<br>$\text{push}(t_m, tk_{out})$ |
| **data operations** | |
| $(y) = \mathsf{Const}(c)$ | $\text{push}(c, y)$ |
| $(y) = \mathsf{MonOp}(f, x)$ | $t_x = \text{get}(x)$<br>$\text{push}(f(t_x), y)$ |
| $(y) = \mathsf{BinOp}(\odot, x_0, x_1)$ | $t_0 = \text{get}(x_0)$<br>$t_1 = \text{get}(x_1)$<br>$\text{push}(t_0 \odot t_1, y)$ |
| $(y) = \mathsf{ITE}(c, x_1, x_0)$ | $t_c = \text{get}(c)$<br>$t_1 = \text{get}(x_1)$<br>$t_0 = \text{get}(x_0)$<br>$\text{if}(t_c)$ then $\text{push}(t_1, y)$<br>else $\text{push}(t_0, y)$ |

## 3 Dataflow Process Networks (DPNs)

A dataflow graph aka dataflow process network (DPN) [60] is a directed graph whose vertices are process nodes and whose edges are FIFO buffers, i.e., queues, storing values that

are also called tokens. A process node can fire if its input buffers have sufficiently many tokens, where 'sufficiently' depends on the particular kind of node (see Table 1). If the node fires, it consumes tokens from its input buffers, and adds the produced tokens to the tails of its output buffers. Table 1 lists the syntax and semantics of all DPN nodes in an operational manner using the following basic operations on buffers:

- $\mathsf{get}(x)$ consumes and returns the head of buffer $x$, i.e., for a nonempty buffer $x = [x_0, \ldots, x_{n-1}]$, it returns the head value $x_0$ and removes $x_0$ from $x$. If $x$ is empty, $\mathsf{get}(x)$ will wait until a value arrives in $x$ that can be returned. Hence, $\mathsf{get}(x)$ is a *blocking read operation* as required by Kahn [57, 58].
- $\mathsf{push}(x, y)$ adds value $x$ to the tail of buffer $y$; i.e., buffer $y = [y_0, \ldots, y_{n-1}]$ becomes $y \cdot x = [y_0, \ldots, y_{n-1}, x]$

Using these read and write operations on buffers, the meaning of the nodes listed in Table 1 should be clear (see [68] for more information and a translation from sequential programs to dataflow graphs using these nodes). We just remark here that all nodes except for the load/store nodes are Kahn nodes since they are based on the above blocking read method and do not hold a local or global state. The load/store nodes access the shared memory mem that is partitioned into segments $\mathsf{mem}[a, \ldots]$ determined by the additional parameter 'a' (which could be an array) of the load/store nodes. In addition to the operands they need for the memory access, the load/store nodes of the same segment are connected by a token chain (input $tk_{in}$ and output $tk_{out}$) to enforce the sequential order given by a program so that races on the memory are avoided.

## 4 Linear Graph Layouts

To generate code from DPNs, we have to compute (1) a schedule, i.e., we have to map the nodes of the DPN to points of time when they shall be executed, and (2) an allocation that maps each node to one of the PUs of the BED machine. As we will see in this section, scheduling is done best by levelizing a DPN, and the PU allocation has to avoid crossings of edges that refer to the same virtual channels as we will prove in the following section. These results are based on a couple of results from graph drawing considered in this section.

### 4.1 Queue Layouts of Graphs

A general method for drawing graphs [18, 71] is to first partition the vertices into levels such that edges only connect vertices of levels $i$ and $j > i$ and then to permute the vertices of each level to minimize the number of edge crossings. In particular, one considers so-called *linear graph layouts* where the vertices of a graph are placed along a single line, i.e., imposing a total order on the nodes such that the edges have to fulfill certain requirements. For instance, for queue layouts [47], we have the following requirements:

**Definition 2** (Queue Layouts [47]). *A $k$-queue layout of a directed or undirected graph $G = (V, E)$ is given by a total ordering[1] $\preceq$ of the vertices $V$ of the graph $G$ and a partition of the edges $E$ of the graph $G$ into $k$ equivalence classes (queues) given by a mapping $\kappa : E \to \{1, \ldots, k\}$ such that crossing edges must not be assigned to the same queue, i.e., for all $(x_1, y_1) \in E$ and all $(x_2, y_2) \in E$, a crossing $x_1 \preceq x_2 \preceq y_2 \preceq y_1$ implies $\kappa(x_1, y_1) \neq \kappa(x_2, y_2)$. The minimal number $k$ needed for a $k$-queue layout is called the queue number of the graph.*

[24, 42, 47] also considers $k$-stack layouts and compares them with $k$-queue layouts which reveals many surprising differences between both. We will however only consider queue layouts in the following. The seminal paper [47] already conjectured that *planar graphs have a bounded queue number* which has just recently been proved [21, 22, 25] after a series of papers [3, 19, 20, 23, 26, 27]. The currently known lower bound is 4 and the currently known best upper bound is 49 for planar graphs. Moreover, [1] proves that the queue number of planar 3-trees has upper bound five and shows that the lower bound is four. Finally, series-parallel graphs, i.e., graphs with tree-width 2, have queue number 3 [79].

*In terms of BED architectures, we interpret $k$-queue layouts as follows: we consider the given graph $G = (V, E)$ as a DPN and use the total ordering of the vertices as a sequential schedule to fire the nodes on a machine with $k$-queues for storing the operand and result values. When the sequential schedule reaches a node, then all its operands (that correspond to the incoming edges) must be found as heads of the $k$ queues so that the node can fire. The obtained result values (that correspond to the outgoing edges) are then added to the tails of the queues.*

### 4.2 Queue Layouts with Given Vertex Orderings

If we consider a given vertex ordering $\preceq$, then major obstacles for queue layouts are $k$-crossings:

**Definition 3** ($k$-Crossings). *Consider a graph $G = (V, E)$ with a total ordering $\preceq$ of its vertices $V$. Any set of $k$ edges $(x_1, y_1), \ldots, (x_k, y_k)$ with $x_1 \preceq x_2 \preceq \ldots \preceq x_{k-1} \preceq x_k$ form a $k$-crossing iff their endpoints are ordered in the opposite way $y_k \preceq y_{k-1} \preceq \ldots \preceq y_2 \preceq y_1$.*

The following theorem reveals that $k$-crossings are the major obstacles for generating $k$-queue layouts:

**Theorem 1** (Layouts with Fixed Variable Ordering [47]). *A graph $G = (V, E)$ with a total ordering $\preceq$ has $k$-queue layout if and only if it does not have a $k$-crossing.*

For a fixed vertex ordering, it is therefore simple to check whether a $k$-queue layout exists: We just have to check whether the graph contains a $k$-crossing:

[1] By the considered ordering $\preceq$ of the vertices, we induce an orientation on the edges of an undirected graph.

**Theorem 2** (Layouts with Fixed Variable Ordering [47]). *Checking whether a graph with a given vertex ordering has a $k$-queue layout can be done in linear time.*

### 4.3 Recognizing 1-Queue Graphs

The execution of a DPN on a $k$-queue machine requires us to determine the vertex ordering, i.e., the sequential schedule, as well as the assignment of the edges to the $k$-queues, i.e., the allocation of PUs. For the case $k = 1$, the PU allocation becomes trivial. Moreover, the code for a 1-queue machine can also be run on a $k$-queue machine for any number $k$ in that we may distribute the work arbitrarily on the PUs. We are therefore interested in 1-queue layouts.

For a 1-queue layout, Definition 2 demands that for any pair of edges $(x_1, y_1)$ and $(x_2, y_2)$ with $x_1 \preceq x_2$, we have to have $y_1 \preceq y_2$. Heath and Rosenberg proved that undirected 1-queue graphs are exactly the arched level-planar graphs which are defined below, and Pemmeraju et.al. [43, 44, 46, 63] showed the same for directed graphs:

**Definition 4** (Arched Level-Planar Graphs [43, 44, 46, 47, 63]). *A graph $G = (V, E)$ is a level-planar graph with $\ell$ levels if there exists a total ordering $\preceq$ on the vertices and a function $\hbar : V \to \{1, \ldots, \ell\}$ such that (1) there is no crossing of edges and (2) $(x, y) \in E$ with $x \preceq y$ implies $\hbar(y) = \hbar(x) + 1$.*

*For each level $l \in \{1, \ldots, \ell\}$, define $b_l$ (bottom) and $t_l$ (top) as the minimal and maximal vertices of level $l$, and let $s_l$ be the first vertex in level $l$ that is adjacent to some vertex in level $l + 1$, or, if there are no edges between levels $l$ and $l + 1$, let $s_l := t_l$. An arch at level $l$ is an edge from $t_l$ to a vertex $j$ with $b_l \preceq j \preceq \min\{s_l, t_l - 1\}$ (where $t_l - 1$ is the predecessor of $t_l$ w.r.t. $\preceq$).*

Arches are edges from the topmost vertex $t_l$ of a level $l$ to a vertex of the same level that occurs before the first vertex of the same level having an edge to the next level [47]. Clearly, arches can be drawn around the graph so that there is no crossing. The following theorem characterizes the graphs with a 1-queue layout as the arched level-planar graphs:

**Theorem 3** (Arched Level-Planar Graphs [43, 44, 46, 47, 63]). *A directed [47] or undirected [43, 44, 46, 63] graph has a 1-queue layout if and only if it is an arched level-planar graph.*

Hence, for checking the existence of a 1-queue layout, we have to check whether the graph is arched level-planar. For undirected graphs, this problem is NP-complete:

**Theorem 4** (Undirected Arched Level-Planar Graphs [47]). *Given an undirected graph, the following problems are NP-complete:*

- *checking whether the graph is arched level-planar*
- *checking whether the graph is level-planar*
- *checking whether the graph has a 1-queue layout*

From the above results, it also follows that determining the queue number of a graph is NP-complete. The situation is different for directed acyclic graphs (DAGs):

**Theorem 5** (Directed Arched Level-Planar Graphs [43, 45, 53, 63]). *Checking whether a DAG is an arched level-planar DAG can be done in linear time. Hence, checking whether a DAG has a 1-queue layout can be done in linear time.*

However, recognizing $k$-queue DAGs is not as simple as recognizing 1-queue DAGs: [45, 63] already proved the NP-completeness of recognizing 4-queue DAGs. Note also the criticism of [53] on the original proof given in [43, 45, 63] and its correction in [53, 54].

Another linear-time algorithm has been determined for the special case of bipartite graphs, i.e., directed graphs with only two levels. Eades et.al. [28–31, 35] present in [29, 30] a linear-time algorithm for drawing bipartite graphs without edge crossings, if possible. Moreover, they show that the problem of minimizing crossings by determining an ordering of one level while fixing the ordering of the other level is NP-complete. Their linear-time algorithm to determine vertex orderings given in [29, 30] is based on *caterpillars* [32, 40]:

**Definition 5** (Caterpillar [29, 30]). *An undirected graph $G = (V, E)$ is a caterpillar if it is a tree with one path called the backbone, which contains all vertices of degree bigger than one.*

**Theorem 6** (Caterpillar [29, 30]). *A bipartite undirected graph is level-planar if and only if it is a collection of caterpillars. A connected bipartite undirected graph is level-planar if and only if it is a caterpillar.*

Hence, checking whether a bipartite graph has a level-planar drawing is done by checking whether it is a set of caterpillars: To this end, we have to determine the backbones of the caterpillars which can be done by simply deleting all vertices of degree 1 (which removes the legs of the caterpillars) and checking that the remaining vertices form a set of paths. According to the theorem above, there is a level-planar drawing iff we find a collection of caterpillars, and that already gives us the level-planar drawing.

The linear-time algorithm for checking whether a given bipartite graph has a level-planar drawing can be extended to further levels by checking the consistency of node orders of the different levels similar to the algorithm using PQ-trees [43, 45, 63] and [53, 54]. An alternative quadratic time testing and embedding algorithm that uses vertex-exchange graphs was developed by Healy and Kuusik [41].

The DPNs obtained from sequential programs are special directed graphs and also special, i.e., quasi-static, DPNs [68]. However, even with the restriction to basic blocks, we may still require more than one PU (and thus more than one queue) for their execution (see Figures 2, 3, and 4). Since S (swap) and SWT (switch) nodes are not used for the translation of basic blocks [68], the only nodes with outdegree $> 1$ in DPNs of basic blocks are D (duplicate) nodes. Hence, the backbone of a caterpillar in a leveled DPN is an interleaving of D nodes $D_1, \ldots, D_k$ at a level $\mathcal{L}_i$ and other nodes $N_0, \ldots, N_k$ with indegree $> 1$ at level $\mathcal{L}_{i+1}$ (see Figure 3).

While checking whether a given bipartite graph has a level-planar drawing can be done in linear time, the minimization of the number of crossings is NP-complete even if an ordering of one level of the bipartite graph is fixed:

**Theorem 7** (Crossing Minimization of Bipartite Graphs)**.** *The following crossing minimization problems are NP-complete:*

- *Two-Level Ordering [37, 51]: Checking whether there are orderings of the vertices in both levels of a bipartite graph such that there are at most c crossings is NP-complete.*
- *One-Level Ordering [29, 31]: Checking whether there is an ordering of the vertices in one level for a fixed ordering of the vertices of the other level of a bipartite graph such that there are at most c crossings is NP-complete.*
- *Planarization [30, 36, 61, 62]: Removing the minimal number of edges from a 2-level graph such that the remaining graph is planar is NP-complete.*

A reduction of the crossing minimization problem to an ILP problem is presented in [55, 56]. Heuristics for reducing crossings are found in [28, 31, 55, 56, 62]. In particular, [31] considers the median and the barycentric heuristics to minimize crossings for the one-level ordering problem.

## 5 BED Code Generation from DPNs

In this section, we explain how to generate move code for BED machines from dataflow graphs. To this end, we use DPNs as described in Section 3 as intermediate representation of the given sequential program [68]. Based on the results of queue layouts of graphs, we use leveled DPNs and have to avoid *certain* edge crossings by choosing suitable node orderings and PU assignments for code generation (see Theorem 8). To discuss which edge crossings of the leveled DPN must be avoided, we first have to determine how the buffers of the DPN nodes are mapped to the buffers of the PUs of the BED machine in Section 5.1. *Section 5.2 contains then a main insight of the paper in that only those edge crossings have to be avoided that refer to the same virtual channel, i.e., to the same move instruction* src → tgt. Sections 5.3 and 5.4 discuss then the need of edge orderings and how these can be derived from node orderings.

### 5.1 Mapping DPN Buffers to PU Buffers

To decide whether an edge crossing in a leveled DPN is a critical one (see Section 5.2), we first have to map the buffers of the DPN nodes to the buffers of the PUs of the BED machine. If we would use only one input and one output buffer for all PUs, the BED machine would become equivalent to a queue machine, and every edge crossing would be a critical one according to Theorem 3. However, we will see in Theorem 8 that the more input and output buffers PUs have, the less conflicts will arise. For instance, we may use the following mapping of the DPN buffers to the input/output buffers of PUs with three input buffers $in_0$,$in_1$,$in_2$ and two output buffers $out_0$,$out_1$:

| node | $in_0$ | $in_1$ | $in_2$ | $out_0$ | $out_1$ |
|---|---|---|---|---|---|
| $(y) := \mathsf{C}(x)$ | $x$ | – | – | $y$ | |
| $(y_0, y_1) := \mathsf{D}(x)$ | $x$ | – | – | $y_0$ | $y_1$ |
| $(y_0, y_1) := \mathsf{S}(x_0, x_1)$ | $x_0$ | $x_1$ | – | $y_0$ | $y_1$ |
| $(y) := \mathsf{J}(x_0, x_1)$ | $x_0$ | $x_1$ | – | $y$ | |
| $() := \mathsf{K}(x)$ | $x$ | – | – | – | |
| $(y) = \mathsf{SEL}(c, x_1, x_0)$ | $c$ | $x_1$ | $x_0$ | $y$ | |
| $(y_1, y_0) = \mathsf{SWT}(c, x)$ | $c$ | $x$ | – | $y_1$ | $y_0$ |
| $(tk_{out}, y) = \mathsf{LD}_a(adr, tk_{in})$ | $adr$ | $tk_{in}$ | – | $tk_{out}$ | $y$ |
| $(tk_{out}) = \mathsf{ST}_a(adr, tk_{in}, x)$ | $adr$ | $x$ | $tk_{in}$ | $tk_{out}$ | |
| $(y) = \mathsf{Const}(c)$ | – | – | – | $y$ | |
| $(y) = \mathsf{MonOp}(f, x)$ | $x$ | – | – | $y$ | |
| $(y) = \mathsf{BinOp}(\odot, x_0, x_1)$ | $x_0$ | $x_1$ | – | $y$ | |
| $(y) = \mathsf{ITE}(c, x_1, x_0)$ | $c$ | $x_1$ | $x_0$ | $y$ | |

Looking at the above table, we see that only SEL, ITE, and $\mathsf{ST}_a$ nodes use all three input buffers. Fortunately, we do not have to consider SEL and SWT nodes in the BED code since these nodes are replaced by branch instructions in the BED code (recall that BED architectures are still sequential processors with a program counter). Hence, we can ignore SEL and SWT nodes. We can also ignore ITE nodes since these can be replaced by other nodes (see [68]).

Finally, since BED programs are sequential programs, they provide a strict ordering of the move code instructions. Clearly, that ordering of the instructions will follow the token chains of the load/store nodes so that there is no longer a need for the access tokens $tk_{in}$ and $tk_{out}$ in the BED code. Note that these access tokens just impose constraints for scheduling the DPN as already done in the scheduling (leveling) phase.

Hence, we actually need at most two input and two output buffers for implementing the DPN nodes in the BED machine. Therefore, we define the following standard PU and buffer mapping for the rest of the paper:

**Definition 6** (Mapping DPN Buffers to PU Buffers)**.** *The standard PU of a BED machine has input buffers* inL, inR, *and* opc *for the left-hand side operand, the right-hand side operand, and the opcode of an instruction, and output buffers* outL *and* outR *for the possible two result values. The input and output buffers of the DPN nodes shown in Table 1 except for nodes* SEL, SW, *and* ITE *are mapped as follows to the PU input and output buffers (recall that array access tokens can be ignored in the BED code):*

| *node* | inL | inR | opc | outL | outR |
|---|---|---|---|---|---|
| $(y) := \mathsf{C}(x)$ | $x$ | – | C | $y$ | – |
| $(y_0, y_1) := \mathsf{D}(x)$ | $x$ | – | D | $y_0$ | $y_1$ |
| $(y_0, y_1) := \mathsf{S}(x_0, x_1)$ | $x_0$ | $x_1$ | S | $y_0$ | $y_1$ |
| $(y) := \mathsf{J}(x_0, x_1)$ | $x_0$ | $x_1$ | J | $y$ | – |
| $() := \mathsf{K}(x)$ | $x$ | – | K | – | – |
| $(tk_{out}, y) = \mathsf{LD}_a(adr, tk_{in})$ | $adr$ | – | LD | $y$ | – |
| $(tk_{out}) = \mathsf{ST}_a(adr, tk_{in}, x)$ | $adr$ | $x$ | ST | – | – |
| $(y) = \mathsf{Const}(c)$ | – | – | CS(c) | $y$ | – |
| $(y) = \mathsf{MonOp}(f, x)$ | $x$ | – | $f$ | $y$ | – |
| $(y) = \mathsf{BinOp}(\odot, x_0, x_1)$ | $x_0$ | $x_1$ | $\odot$ | $y$ | – |

Note that if more than one DPN buffer would have to be mapped to the same PU buffer, the consumption/production order as specified in Table 1 has to be respected. For instance, if ITE nodes shall be used, we may map $c$ to inL and $x_1, x_0$ to inR (in that order).

As we will see by Theorem 8, having two input and two output buffers for operands and results allows us to resolve many crossings without the need of swap nodes or the need for further PUs.

### 5.2 Critical Crossings in Leveled DPNs

According to Theorem 5, we have to resolve *all* crossings in a leveled DPN to be able to generate code for a 1-queue machine. However, if PUs have more than one input or more than one output buffer, it turns out that *not all crossings are critical*. To understand which crossings are critical, recall that we generate the move code from a leveled DPN by a level-by-level left-to-right traversal. When reaching a node, we assume that the inputs are found as the heads of the input buffers so that the node can fire at this point of time. Also, we may assume that the output values are produced at this point of time.

Consider two DPN buffers (or edges) $p_1 \longrightarrow q_1$ and $p_2 \longrightarrow q_2$ with $p_1 \prec p_2$ and $q_2 \prec q_1$ so that they form a crossing. Assume that $p_1 \longrightarrow q_1$ and $p_2 \longrightarrow q_2$ correspond by the chosen buffer mapping to move instructions $\mathrm{src}_1 \longrightarrow \mathrm{tgt}_1$ and $\mathrm{src}_2 \longrightarrow \mathrm{tgt}_2$ on the BED machine, respectively. Finally, assume that these moves transport the values $v_1, v_2$ produced by the nodes $p_1, p_2$, respectively. Due to the level-by-level left-to-right schedule, value $v_1$ is produced by $p_1$ and put in output buffer $\mathrm{src}_1$ before the value $v_2$ is produced by $p_2$ and put in output buffer $\mathrm{src}_2$. Moreover, value $v_2$ is consumed by $q_2$ before value $v_1$ is consumed by $q_1$ in the next level due to our node schedule. Ordering the move instructions in the BED program depends on certain cases as discussed below:

1. If $\mathrm{src}_1 \neq \mathrm{src}_2$ and $\mathrm{tgt}_1 \neq \mathrm{tgt}_2$ holds, then the two moves are independent of each other and can be issued in any order. A crossing of such edges is uncritical, since the head elements of different buffers become tail elements of different buffers.
2. If $\mathrm{src}_1 \neq \mathrm{src}_2$ and $\mathrm{tgt}_1 = \mathrm{tgt}_2$ holds, then the values $v_1, v_2$ are found in different output buffers and have to be moved to the same input buffer $\mathrm{tgt} := \mathrm{tgt}_1 = \mathrm{tgt}_2$. As in the next level, $v_2$ is expected before $v_1$ in this input buffer tgt, we must issue first $\mathrm{src}_2 \longrightarrow \mathrm{tgt}_2$ and then $\mathrm{src}_1 \longrightarrow \mathrm{tgt}_1$ to ensure the required consumption order. Note that this is possible since $\mathrm{src}_1 \neq \mathrm{src}_2$ holds so that we can access the values $v_1, v_2$ in any order even though $v_1$ has been produced before $v_2$.
3. If $\mathrm{src}_1 = \mathrm{src}_2$ and $\mathrm{tgt}_1 \neq \mathrm{tgt}_2$ holds, then the values $v_1, v_2$ have to be moved from the same output buffer $\mathrm{src} := \mathrm{src}_1 = \mathrm{src}_2$ to different input buffers $\mathrm{tgt}_1 \neq \mathrm{tgt}_2$. Due to our node schedule, $v_1$ occurs before $v_2$ in the output buffer src. We therefore have to put first $\mathrm{src}_1 \longrightarrow \mathrm{tgt}_1$ and then $\mathrm{src}_2 \longrightarrow \mathrm{tgt}_2$ in the move code program. Note that this is no problem even if $v_2$ is consumed before $v_1$ in the next level, since the values become tails of different input buffers $\mathrm{tgt}_1 \neq \mathrm{tgt}_2$.
4. If $\mathrm{src}_1 = \mathrm{src}_2$ and $\mathrm{tgt}_1 = \mathrm{tgt}_2$ holds, then the values $v_1, v_2$ have to be moved from the same output buffer $\mathrm{src} := \mathrm{src}_1 = \mathrm{src}_2$ to the same input buffer $\mathrm{tgt} := \mathrm{tgt}_1 \neq \mathrm{tgt}_2$. Due to the schedule, $v_1$ has been produced and put in output buffer src before $v_2$. We cannot handle this case without a swap operation since we have to move $v_1$ first since it is the head of the output buffer src, but we need to consume $v_2$ first from buffer tgt.

We therefore have proved the following theorem:

**Theorem 8** (Crossings of Virtual Channels)**.** *Consider two crossing DPN edges $v_1 : p_1 \longrightarrow q_1$ and $v_2 : p_2 \longrightarrow q_2$ with $p_1 \prec p_2$ and $q_2 \prec q_1$ that correspond by the mapping of DPN buffers to PU buffers to the move instructions $\mathrm{src}_1 \longrightarrow \mathrm{tgt}_1$ and $\mathrm{src}_2 \longrightarrow \mathrm{tgt}_2$, respectively. Then, the following holds*

- *If $\mathrm{src}_1 \neq \mathrm{src}_2$, and $\mathrm{tgt}_1 \neq \mathrm{tgt}_2$ holds, then every order of $\mathrm{src}_1 \longrightarrow \mathrm{tgt}_1$ and $\mathrm{src}_2 \longrightarrow \mathrm{tgt}_2$ in the move code is correct.*
- *If $\mathrm{src}_1 = \mathrm{src}_2$, and $\mathrm{tgt}_1 \neq \mathrm{tgt}_2$ holds, then $\mathrm{src}_1 \longrightarrow \mathrm{tgt}_1$ must occur before $\mathrm{src}_2 \longrightarrow \mathrm{tgt}_2$ in the move code.*
- *If $\mathrm{src}_1 \neq \mathrm{src}_2$, and $\mathrm{tgt}_1 = \mathrm{tgt}_2$ holds, then $\mathrm{src}_2 \longrightarrow \mathrm{tgt}_2$ must occur before $\mathrm{src}_1 \longrightarrow \mathrm{tgt}_1$ in the move code.*
- *If $\mathrm{src}_1 = \mathrm{src}_2$, and $\mathrm{tgt}_1 = \mathrm{tgt}_2$ holds, then there is no correct move code sequence.*

*The above theorem therefore derives from a given node ordering constraints for edge orderings, i.e., constraints for ordering move instructions in the move code. Moreover, note that even if the DPN edges $v_1 : p_1 \longrightarrow q_1$ and $v_2 : p_2 \longrightarrow q_2$ do not have a crossing, i.e., $p_1 \prec p_2$ and $q_1 \preceq q_2$ holds, the move code sequence must still be consistent with the node schedule.* That is, if the edges are mapped to the same output (resp. input) buffers in the BED machine, the move instructions should be appropriately ordered to dequeue (resp. enqueue) values $v_1, v_2$ in that order following the node schedule.

Since a 1-queue machine has only one virtual channel, *all* crossings are critical. For a $k$-queue machine with $k > 1$ and a $k$-BED machine with $k \geq 1$ where PUs have more than one input or more than one output buffer, many crossings are uncritical for move code generation.

### 5.3 Deriving Edge Orderings from Node Orderings

*For a given node ordering, we have to make sure that the constraints of Theorem 8 can be satisfied by a suitable edge ordering, i.e., a move code program.* Given a node ordering, it is not difficult to determine a related edge ordering (or move instruction ordering) by a kind of symbolic simulation: To that end, we compute for every input and output buffer of every PU the stream of values that will pass through that buffer. This is done by ordering the DPN nodes and selecting the operands and results associated with the buffers.

```
nat x00,x01,x02,x03;
nat x30,x31,x32,x33;

thread MinTwoPUsNeeded {
    nat x10,x11,x12,x13;
    nat x20,x21,x22,x23;

    x10 = x00 + x01;
    x11 = x01 + x03;
    x12 = x02 + x00;
    x13 = x03 + x02;

    x20 = x10 + x13;
    x21 = x11 + x11;
    x22 = x12 + x12;
    x23 = x13 + x10;

    x30 = x20 + x23;
    x31 = x21 + x22;
    x32 = x22 + x21;
    x33 = x23 + x20;
}
```

```
DPN MinTwoPUsNeeded {
 inputs:  x00,x01,x02,x03;
 outputs: x30,x31,x32,x33;
 process nodes:
  N00: (v00,w00) := Dup(x00);
  N01: (v01,w01) := Dup(x01);
  N02: (v02,w02) := Dup(x02);
  N03: (v03,w03) := Dup(x03);

  N04: (x10) := AddN(v00,v01);
  N05: (x11) := AddN(w01,v03);
  N06: (x12) := AddN(v02,w00);
  N07: (x13) := AddN(w03,w02);

  N08: (v10,w10) := Dup(x10);
  N09: (v11,w11) := Dup(x11);
  N10: (v12,w12) := Dup(x12);
  N11: (v13,w13) := Dup(x13);

  N12: (x20) := AddN(v10,v13);
  N13: (x21) := AddN(v11,w11);
  N14: (x22) := AddN(v12,w12);
  N15: (x23) := AddN(w13,w10);

  N16: (v20,w20) := Dup(x20);
  N17: (v21,w21) := Dup(x23);
  N18: (v22,w22) := Dup(x21);
  N19: (v23,w23) := Dup(x22);

  N20: (x30) := AddN(v20,v21);
  N21: (x31) := AddN(v22,v23);
  N22: (x32) := AddN(w23,w22);
  N23: (x33) := AddN(w21,w20);
}
```

**Figure 2.** Program `MinTwoPUsNeeded` and its DPN

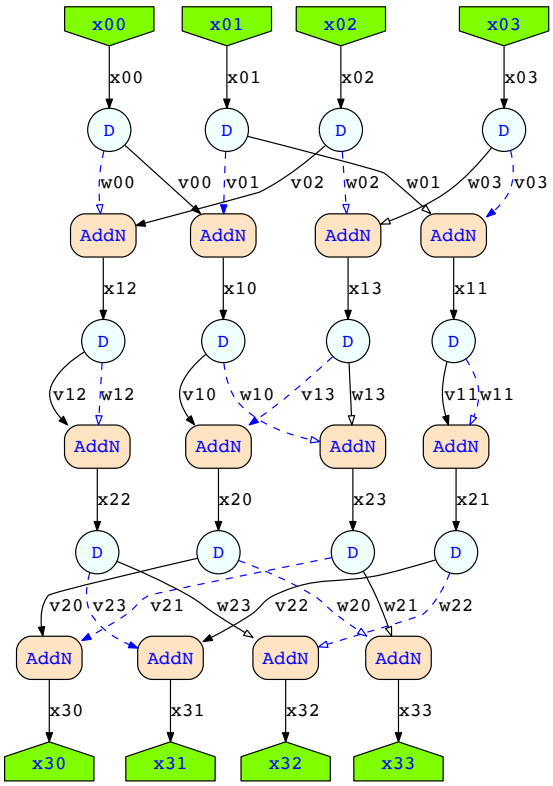


**Figure 3.** Dataflow Graph for Program `MinTwoPUsNeeded`

For example, consider the DPN shown in Figures 2 and 3 that we will schedule on two PUs[2] such that the nodes with even and odd indices are scheduled to `PU[0]` and `PU[1]`, respectively. The nodes are simply ordered by their indices in this example.

Scanning the nodes from top to bottom (following the node ordering), we extract the following streams for the buffers of the PUs:

```
PU[0]inL :x00,x02,v00,v02,x10,x12,v10,v12,x20,x21,v20,w23
PU[0]inR :v01,w00,v13,w12,v21,w22
PU[0]outL:v00,v02,x10,x12,v10,v12,x20,x22,v20,v22,x30,x32
PU[0]outR:w00,w02,w10,w12,w20,w22
PU[1]inL :x01,x03,w01,w03,x11,x13,v11,w13,x23,x22,v22,w21
PU[1]inR :v03,w02,w11,w10,v23,w20
PU[1]outL:v01,v03,x11,x13,v11,v13,x21,x23,v21,v23,x31,x33
PU[1]outR:w01,w03,w11,w13,w21,w23
```

From these streams, it is straightforward to generate the move code program: We just have to check during a symbolic execution which output and input buffers have the same heads and whenever two heads match, we may issue that move instruction (inputs and outputs can be moved directly). For this example, we therefore obtain the move program shown in Figure 4.

[2]With the SAT constraints of Definition 7, we can prove that at least two PUs are actually required for scheduling this DPN.

### 5.4 Existence of Edge Orderings

Section 5.3 explains how to generate move code for a BED machine from a DPN with a given node ordering (if possible) such that the constraints of Theorem 8 are satisfied. *Note that each edge crossing may impose an ordering constraint for the corresponding move code instructions, and we have to satisfy all of these constraints at the end. We therefore also have to enforce the existence of edge orderings.*

As we will demonstrate with the example below, it is thereby necessary that the constraints on the ordering of the move code are consistently embedded in an edge ordering. To see this, consider the following example program on the left that is translated to the DPN on the right-hand side below:

```
nat x0,x1,x2,x3;
nat y0,y1,y2,y3;

thread bufOrdNeed {
    y0 = x0 + x2;
    y1 = x1 + x3;
    y2 = x2 + x1;
    y3 = x3 + x0;
}
```

```
DPN bufOrdNeed {
    inputs:  x0,x1,x2,x3;
    outputs: y0,y1,y2,y3;
    process nodes:
        p2: (v2,w2) := Dup(x2);
        p0: (v0,w0) := Dup(x0);
        p1: (v1,w1) := Dup(x1);
        p3: (v3,w3) := Dup(x3);
        q2: (y2) := AddN(v2,v1);
        q1: (y1) := AddN(w1,v3);
        q0: (y0) := AddN(v0,w2);
        q3: (y3) := AddN(w3,w0);
}
```

```
x00 -> PU[0].inL // move x00
x02 -> PU[0].inL // move x02
x01 -> PU[1].inL // move x01
x03 -> PU[1].inL // move x03
Dup -> PU[0].opc // fire N00: (v00,w00) := Dup(x00)
Dup -> PU[1].opc // fire N01: (v01,w01) := Dup(x01)
Dup -> PU[0].opc // fire N02: (v02,w02) := Dup(x02)
Dup -> PU[1].opc // fire N03: (v03,w03) := Dup(x03)
PU[0].outL -> PU[0].inL // move v00
PU[0].outL -> PU[0].inL // move v02
PU[1].outL -> PU[0].inR // move v01
PU[0].outR -> PU[0].inR // move w00
PU[1].outR -> PU[1].inL // move w01
PU[1].outR -> PU[1].inL // move w03
PU[1].outL -> PU[1].inR // move v03
PU[0].outR -> PU[1].inR // move w02
AddN -> PU[0].opc // fire N04: (x10) := AddN(v00,v01)
AddN -> PU[1].opc // fire N05: (x11) := AddN(w01,v03)
AddN -> PU[0].opc // fire N06: (x12) := AddN(v02,w00)
AddN -> PU[1].opc // fire N07: (x13) := AddN(w03,w02)
PU[0].outL -> PU[0].inL // move x10
PU[0].outL -> PU[0].inL // move x12
PU[1].outL -> PU[1].inL // move x11
PU[1].outL -> PU[1].inL // move x13
Dup -> PU[0].opc // fire N08: (v10,w10) := Dup(x10)
Dup -> PU[1].opc // fire N09: (v11,w11) := Dup(x11)
Dup -> PU[0].opc // fire N10: (v12,w12) := Dup(x12)
Dup -> PU[1].opc // fire N11: (v13,w13) := Dup(x13)
PU[0].outL -> PU[0].inL // move v10
PU[0].outL -> PU[0].inL // move v12
PU[1].outL -> PU[1].inL // move v11
PU[1].outR -> PU[1].inR // move w11
PU[0].outR -> PU[1].inR // move w10
PU[1].outL -> PU[0].inR // move v13
PU[0].outR -> PU[0].inR // move w12
PU[1].outR -> PU[1].inL // move w13
AddN -> PU[0].opc // fire N12: (x20) := AddN(v10,v13)
AddN -> PU[1].opc // fire N13: (x21) := AddN(v11,w11)
AddN -> PU[0].opc // fire N14: (x22) := AddN(v12,w12)
AddN -> PU[1].opc // fire N15: (x23) := AddN(w13,w10)
PU[0].outL -> PU[0].inL // move x20
PU[1].outL -> PU[0].inL // move x21
PU[1].outL -> PU[1].inL // move x23
PU[0].outL -> PU[1].inL // move x22
Dup -> PU[0].opc // fire N16: (v20,w20) := Dup(x20)
Dup -> PU[1].opc // fire N17: (v21,w21) := Dup(x23)
Dup -> PU[0].opc // fire N18: (v22,w22) := Dup(x21)
Dup -> PU[1].opc // fire N19: (v23,w23) := Dup(x22)
PU[0].outL -> PU[0].inL // move v20
PU[1].outL -> PU[0].inR // move v21
PU[0].outL -> PU[1].inL // move v22
PU[1].outR -> PU[1].inL // move w21
PU[1].outL -> PU[1].inR // move v23
PU[0].outR -> PU[1].inR // move w20
PU[1].outR -> PU[0].inL // move w23
PU[0].outR -> PU[0].inR // move w22
AddN -> PU[0].opc // fire N20: (x30) := AddN(v20,v21)
AddN -> PU[1].opc // fire N21: (x31) := AddN(v22,v23)
AddN -> PU[0].opc // fire N22: (x32) := AddN(w23,w22)
AddN -> PU[1].opc // fire N23: (x33) := AddN(w21,w20)
PU[0].outL -> x30 // move x30
PU[0].outL -> x32 // move x32
PU[1].outL -> x31 // move x31
PU[1].outL -> x33 // move x33
```

**Figure 4.** Move Code for Program `MinTwoPUsNeeded`

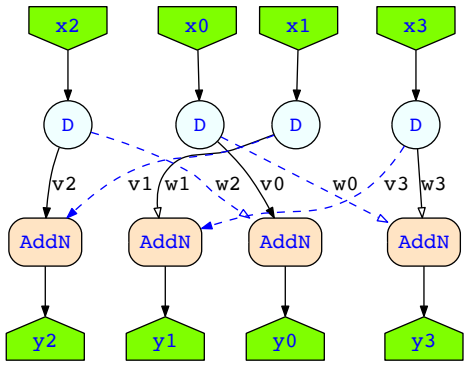


**Figure 5.** DPN `bufOrdNeeded` has no critical crossing.

Figure 5 shows the graphical representation of the DPN with the following node ordering:

$$\texttt{p2} \prec \texttt{p0} \prec \texttt{p1} \prec \texttt{p3} \prec \texttt{q2} \prec \texttt{q1} \prec \texttt{q0} \prec \texttt{q3}$$

Mapping all nodes to the same PU, we have to make sure that all crossing edges refer to different virtual channels by Theorem 8. By checking all pairs of buffers, we find the following nine crossings that are all uncritical, i.e., we have $\text{src}_1 \neq \text{src}_2$ or $\text{tgt}_1 \neq \text{tgt}_2$ in each case:

| DPN buffer | VC | × | DPN buffer | VC |
|---|---|---|---|---|
| | $\text{src}_1 \rightarrow \text{tgt}_1$ | × | | $\text{src}_2 \rightarrow \text{tgt}_2$ |
| v0 : p0 → q0 | outL → inL | × | w1 : p1 → q1 | outR → inL |
| v0 : p0 → q0 | outL → inL | × | v3 : p3 → q1 | outL → inR |
| v0 : p0 → q0 | outL → inL | × | v1 : p1 → q2 | outL → inR |
| w2 : p2 → q0 | outR → inR | × | w1 : p1 → q1 | outR → inL |
| w2 : p2 → q0 | outR → inR | × | v3 : p3 → q1 | outL → inR |
| w2 : p2 → q0 | outR → inR | × | v1 : p1 → q2 | outL → inR |
| w0 : p0 → q3 | outR → inR | × | w1 : p1 → q1 | outR → inL |
| w0 : p0 → q3 | outR → inR | × | v3 : p3 → q1 | outL → inR |
| w0 : p0 → q3 | outR → inR | × | v1 : p1 → q2 | outL → inR |

*Still, we cannot generate move code as discussed below. Theorem 8 gives us ordering constraints for the edges that are fulfilled in the above example. However, the 'ordering' of edges must really be an ordering, i.e., it must be transitive, and acyclic which is not possible in this example:* Computing the streams as discussed in the previous section yields:

```
PU[0].inL  : x2,x0,x1,x3,v2;w1;v0;w3
PU[0].inR  : v1;v3;w2;w0
PU[0].outL : v2;v0;v1;v3,y2,y1,y0,y3
PU[0].outR : w2;w0;w1;w3
```

Clearly, we may first move the input values to `inL`, and can then fire the D nodes `p2`, `p0`, `p1`, `p3` to finish the first level. After this, we can still move `v2` from `PU[0].outL` to `PU[0].inL`, but then, we get stuck with the following situation where all heads of the input and output buffers differ:

```
PU[0].inL  : w1;v0;w3
PU[0].inR  : v1;v3;w2;w0
PU[0].outL : v0;v1;v3,y2,y1,y0,y3
PU[0].outR : w2;w0;w1;w3
```

Hence, we cannot generate move code for this node ordering although we satisfy the crossing constraints of Theorem 8. To understand why this is the case, consider the following constraints for the edge ordering $\sqsubset$ that are derived from the nine crossings in Figure 5 according to Theorem 8:

| | | |
|---|---|---|
| $v0 \times w1 \rightsquigarrow w1 \sqsubset v0$ | $v0 \times v3 \rightsquigarrow v0 \sqsubset v3$ | $v0 \times v1 \rightsquigarrow v0 \sqsubset v1$ |
| $w2 \times w1 \rightsquigarrow w2 \sqsubset w1$ | $w2 \times v3 \rightsquigarrow v3 \sqsubset w2$ | $w2 \times v1 \rightsquigarrow v1 \sqsubset w2$ |
| $w0 \times w1 \rightsquigarrow w0 \sqsubset w1$ | $w0 \times v3 \rightsquigarrow v3 \sqsubset w0$ | $w0 \times v1 \rightsquigarrow v1 \sqsubset w0$ |

The above constraints on the edge ordering preclude the generation of move code since they induce a cyclic relation: We have the following four cycles

- $w1 \sqsubset v0 \sqsubset v3 \sqsubset w0 \sqsubset w1$
- $w1 \sqsubset v0 \sqsubset v1 \sqsubset w0 \sqsubset w1$
- $w1 \sqsubset v0 \sqsubset v3 \sqsubset w2 \sqsubset w1$
- $w1 \sqsubset v0 \sqsubset v1 \sqsubset w2 \sqsubset w1$

*We therefore have to make sure that the constraints of Theorem 8 are satisfied by a suitable edge ordering, i.e., an* ***ordering relation*** *that satisfies these constraints. There are node orderings where we can satisfy the constraints of Theorem 8 but cannot find a consistent edge ordering that will do so.*

## 6 Computing Node and Edge Orderings

The previous section makes clear that BED code generation requires us to determine a node ordering with a related edge ordering such that these orderings satisfy the crossing constraints of Theorem 8. In this section, we reduce this problem to a SAT problem (since it is NP-complete) such that SAT solvers can determine the required orderings for a given number of PUs (if possible). Compared to other SAT or ILP encodings like [5, 7, 17, 36], our results about virtual channels given in Theorem 8 greatly simplify the SAT encoding as shown below:

**Definition 7** (SAT Constraints)**.** *Consider a leveled DPN with nodes $\mathcal{P} := \bigcup_{i=1}^{m} \mathcal{L}_i$, disjoint levels $\mathcal{L}_i \cap \mathcal{L}_j = \{\}$, and buffers $\mathcal{B}$ where each buffers $b : p \to q$ connects nodes of successive levels, i.e., $p \in \mathcal{L}_i$ implies $q \in \mathcal{L}_{i+1}$. The following constraints using propositional variables $\alpha_{p,k}$ that hold when DPN node $p$ is assigned to PU $k$, a strict ordering relation $\prec$ on the nodes, and a strict ordering relation $\sqsubset$ on the edges, encode the schedulability problem for $\varrho$ PUs:*

- *existence of a strict DPN node ordering $\prec$:*
  - *irreflexivity: $\neg p \prec p$ for all nodes $p$*
  - *transitivity: for all levels $\mathcal{L}_i$ and nodes $p_1, p_2, p_3 \in \mathcal{L}_i$, we add $p_1 \prec p_2 \wedge p_2 \prec p_3 \to p_1 \prec p_3$*
  - *level totality: for all levels $\mathcal{L}_i$ and nodes $p_1, p_2 \in \mathcal{L}_i$ with $p_1 \neq p_2$, we add $p_1 \prec p_2 \vee p_2 \prec p_1$*
- *existence of a strict DPN edge ordering $\sqsubset$:*
  - *irreflexivity: $\neg b \sqsubset b$ for all buffers $b$*
  - *transitivity: for all pairs $\mathcal{L}_i, \mathcal{L}_{i+1}$ of levels, and buffers $b_1, b_2, b_3$ between $\mathcal{L}_i$ and $\mathcal{L}_{i+1}$, we add the constraint $b_1 \sqsubset b_2 \wedge b_2 \sqsubset b_3 \to b_1 \sqsubset b_3$*
  - *level totality: for all pairs $\mathcal{L}_i, \mathcal{L}_{i+1}$ of levels, and buffers $b_1, b_2$ between $\mathcal{L}_i$ and $\mathcal{L}_{i+1}$ with $b_1 \neq b_2$, we add $b_1 \sqsubset b_2 \vee b_2 \sqsubset b_1$*
- *PU assignment*
  - *all DPN nodes $p \in \mathcal{P}$ are assigned to a PU $1, \ldots, \varrho$:*

$$\bigwedge_{p \in \mathcal{P}} \bigvee_{k=1}^{\varrho} \alpha_{p,k}$$

  - *DPN nodes are assigned to no more than one PU:*

$$\bigwedge_{p \in \mathcal{P}} \bigwedge_{k=1}^{\varrho} \left( \alpha_{p,k} \to \bigwedge_{j=1 \neq k}^{\varrho} \neg \alpha_{p,j} \right)$$

- *crossing constraints: for all pairs $\mathcal{L}_i, \mathcal{L}_{i+1}$ of levels, edges $b_1 : p_1 \to q_1$ and $b_2 : p_2 \to q_2$ with $p_1, p_2 \in \mathcal{L}_i$ and $q_1, q_2 \in \mathcal{L}_{i+1}$, we define (the functions $\mathsf{inBf}(b)$ and $\mathsf{outBf}(b)$ determine which input or output port of a PU is used by buffer $b$ according to Definition 6):*
  - $\mathsf{samePU}(p_1, p_2) :\Leftrightarrow \bigvee_{k=1}^{\varrho} \alpha_{p_1,k} \wedge \alpha_{p_2,k}$
  - $\mathsf{srcEQ} :\Leftrightarrow \mathsf{outBf}(b_1) = \mathsf{outBf}(b_2) \wedge \mathsf{samePU}(p_1, p_2)$
  - $\mathsf{tgtEQ} :\Leftrightarrow \mathsf{inBf}(b_1) = \mathsf{inBf}(b_2) \wedge \mathsf{samePU}(q_1, q_2)$

  *Using the above, we then define the following constraints to ensure that the edge ordering $\sqsubset$ is consistent with the node ordering $\prec$:*
  - $p_1 \prec p_2 \wedge \mathsf{srcEQ} \to b_1 \sqsubset b_2$
  - $q_2 \prec q_1 \wedge \mathsf{tgtEQ} \to b_2 \sqsubset b_1$

The crossing constraints essentially specify the FIFO behavior of the buffers: First, the constraint $p_1 \prec p_2 \wedge \mathsf{srcEQ} \to b_1 \sqsubset b_2$ specifies that if tokens[3] $b_1, b_2$ are produced and written to the same FIFO buffer in that order, it is required that the move instructions are ordered as $b_1 \sqsubset b_2$ so that $b_1$ is moved out of this buffer before $b_2$. Second, $q_2 \prec q_1 \wedge \mathsf{tgtEQ} \to b_2 \sqsubset b_1$ specifies that if tokens $b_2, b_1$ are consumed from the same FIFO buffer in that order, it is required that the move instructions are ordered as $b_2 \sqsubset b_1$ so that $b_2$ is moved into this buffer before $b_1$.

Note also that the latter constraints imply the following condition which is the unsolvable case in Theorem 8:

$$\neg(\underbrace{p_1 \prec p_2 \wedge q_2 \prec q_1}_{\text{cross}} \wedge \mathsf{srcEQ} \wedge \mathsf{tgtEQ})$$

Moreover, note that $p_1 \prec p_2 \wedge \mathsf{srcEQ} \to b_1 \sqsubset b_2$ and $q_2 \prec q_1 \wedge \mathsf{tgtEQ} \to b_2 \sqsubset b_1$ are equivalent to $\mathsf{srcEQ} \to (p_1 \prec p_2 \leftrightarrow b_1 \sqsubset b_2)$ and $\mathsf{tgtEQ} \to (q_2 \prec q_1 \leftrightarrow b_2 \sqsubset b_1)$, respectively. *Hence, node orderings and edge orderings determine each other.*

Finally, note that we do not care about the ordering of nodes of different levels in the SAT constraints; they may be arbitrarily ordered or may not be in ordering relation at all.

A solution to the above SAT constraints yields a strict total ordering for the nodes of each level, a strict total ordering of the buffers between the levels, and an assignment of the nodes to the PUs such that we can generate a BED program. The converse is also the case:

[3] Note that we can identify tokens (values) $b_i$ with their unique move instruction $b_i : p_i \to q_i$ in that we use the same name for both.

**Theorem 9.** *Every move code program for a BED machine satisfies the SAT constraints of Definition 7, and any solution of the SAT constraints of Definition 7 yields a correct BED program. In particular, the level-wise arrangement of the edge ordering directly yields the move code program.*

*Proof.* Clearly, the SAT constraints are all necessary, i.e., any move code program will satisfy them. For the converse implication, we have to prove that if the SAT constraints are met, then also a move program can be derived. Assume, on the contrary, that a node ordering $\prec$, an edge ordering $\sqsubset$, and an assignment of nodes of a leveled DPN to PUs of a BED machine satisfies the SAT constraints of Definition 7, but does not yield a correct move program. That is, the move code generation according to Section 5.4 gets stuck in a situation where all heads of input and output buffers differ.

In this situation, assume that b1 is at the head of some input buffer, w.l.o.g `in[0]`. Then b1 must also be a part of some output buffer, w.l.o.g `out[0]`, however not at its head. Assume that b2 is at the head of `out[0]` as shown in the streams in the left column below.

| | |
|---|---|
| `in[0] : b1;..;..` | `in[0] : b1;..;..` |
| `in[1] : ..;..;..` | `in[1] : b3;..;b2` |
| `in[2] : ..;..;..` | `in[2] : ..;..;..` |
| `:` | `:` |
| `out[0] : b2;..;b1` | `out[0] : b2;..;b1` |
| `out[1] : ..;..;..` | `out[1] : ..;..;..` |
| `out[2] : ..;..;..` | `out[2] : ..;..;..` |
| `:` | `:` |

Clearly, b2 must also be in some input buffer, but not at the head. If b2 would be behind b1 in input buffer `in[0]`, then the edges $p_1 \rightarrow q_1$ and $p_2 \rightarrow q_2$ in the DPN corresponding to buffers b1 and b2 will cross and srcEQ $\wedge$ tgtEQ will hold for b1 and b2 since they share the same virtual channel out[0] $\rightarrow$ in[0]. This reduces the crossing constraint to false which therefore contradicts our assumption that the SAT constraints of Definition 7 were satisfied.

Therefore, b2 must be in some input buffer other than `in[0]`. W.l.o.g assume that b2 is in input buffer `in[1]` whose head is b3 as shown in the right column above. Again, b3 must also occur in some output buffer, but not at the head. If b3 would be behind b2 in output buffer `out[0]`, then the edges corresponding to b2 and b3 would cross and srcEQ $\wedge$ tgtEQ will hold for b2 and b3 since they would share the same virtual channel out[0] $\rightarrow$ in[1]. Again, the crossing constraint would then reduce to false which would contradict our assumption that the SAT constraints of Definition 7 are satisfied. Therefore, b3 must be in some output buffer other than `out[0]`. W.l.o.g assume that b3 is in the output buffer `out[1]` with b4 at its head as shown in the left column below.

| | |
|---|---|
| `in[0] : b1;..;..` | `in[0] : b1;..;..` |
| `in[1] : b3;..;b2` | `in[1] : b3;..;b2` |
| `in[2] : ..;..;..` | `in[2] : b5;..;b4` |
| `:` | `:` |
| `out[0] : b2;..;b1` | `out[0] : b2;..;b1` |
| `out[1] : b4;..;b3` | `out[1] : b4;..;b3` |
| `out[2] : ..;..;..` | `out[2] : ..;..;..` |
| `:` | `:` |

Again, b4 must also be in some input buffer, but not at the head. Arguing similarly as above, we conclude that the crossing constraint excludes that b4 is behind b3 in `in[1]`. Therefore, b4 must be in some input buffer other than `in[1]`.

Now, assume that b4 is behind b1 in `in[0]`. Then, the edge ordering b4 $\sqsubset$ b3 (from `out[1]`), b3 $\sqsubset$ b2 (from `in[1]`), b2 $\sqsubset$ b1 (from `out[0]`), and b1 $\sqsubset$ b4 (from `in[0]`) would close a cycle b4 $\sqsubset$ b3 $\sqsubset$ b2 $\sqsubset$ b1 $\sqsubset$ b4 which contradicts the strict DPN edge ordering constraint in Definition 7. Thus, b4 must be in some input buffer other than `in[0]` and `in[1]`. W.l.o.g, we may assume that b4 is in input buffer `in[2]` with b5 at its head as shown above.

Clearly, we can continue the same line of argumentation until no more buffers are left in the BED processor, at which point we cannot find a stream for the final buffer without contradicting the SAT constraints. This proves that if the SAT constraints are satisfied, then the move code generation according to Section 5.4 must successfully yield a correct BED program. □

## 7 Conclusions

Buffered exposed datapath (BED) architectures are novel processor architectures that consist of a network of PUs whose I/O ports are endowed with FIFO buffers to avoid an unnecessary synchronization of the PUs. This paper suggests to use DPNs as intermediate representation of programs for code generation for these BED architectures. Inspired by results on queue layouts in graph drawing, the paper refines these results and points out their use for code generation from DPNs for BED architectures. In particular, a given DPN first has to be leveled, so that all incoming edges of each node stem from producer nodes of the previous level. This is equivalent to scheduling the DPN nodes for a fully parallel execution schedule where all nodes of a level are executed by different PUs in parallel. If less PUs are to be used, an ordering of the nodes is required that satisfies the criteria of Theorem 8 and Definition 7 for the virtual channels of the BED architecture. Implementations of the code generator using a SAT solver can be found on our web pages[4] and in the corresponding artifact http://doi.org/10.1145/3462321.

[4] https://es.cs.uni-kl.de/tools/teaching/ScadCodeGen.html and https://es.cs.uni-kl.de/tools/teaching/MiniC.html

## References


[1] J.M. Alam, M.A. Bekos, M. Gronemann, M. Kaufmann, and S. Pupyrev. 2020. Queue Layouts of Planar 3-Trees. *Algorithmica* 82, 9 (2020), 2564–2585.

[2] M. Anders, A. Bhagyanath, and K. Schneider. 2018. On Memory Optimal Code Generation for Exposed Datapath Architectures with Buffered Processing Units. In *Application of Concurrency to System Design (ACSD)*, T. Chatain and R. Grosu (Eds.). IEEE Computer Society, Bratislava, Slovakia, 115–124.

[3] M.A. Bekos, H. Förster, M. Gronemann, T. Mchedlidze, F. Montecchiani, C. Raftopoulou, and T. Ueckerdt. 2019. *Planar Graphs of Bounded Degree have Constant Queue Number.* Technical Report arXiv:1811.00816. arXiv.org.

[4] A. Bhagyanath. 2020. *Code Generation for Synchronous Control Asynchronous Dataflow Architectures.* Ph.D. Dissertation. Department of Computer Science, University of Kaiserslautern, Germany.

[5] A. Bhagyanath and K. Schneider. 2016. Optimal Compilation for Exposed Datapath Architectures with Buffered Processing Units by SAT Solvers. In *Formal Methods and Models for Codesign (MEMOCODE)*, E. Leonard and K. Schneider (Eds.). IEEE Computer Society, Kanpur, India, 143–152.

[6] A. Bhagyanath and K. Schneider. 2017. Exploring Different Execution Paradigms in Exposed Datapath Architectures with Buffered Processing Units. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, Y. Patt and S.K. Nandy (Eds.). IEEE Computer Society, Samos, Greece, 1–10.

[7] A. Bhagyanath and K. Schneider. 2017. Exploring the Potential of Instruction-Level Parallelism of Exposed Datapath Architectures with Buffered Processing Units. In *Application of Concurrency to System Design (ACSD)*, A. Legay and K. Schneider (Eds.). IEEE Computer Society, Zaragoza, Spain, 106–115.

[8] G. Blake, R.G. Dreslinski, and T. Mudge. 2009. A Survey of Multicore Processors. *IEEE Signal Processing Magazine* 26, 6 (Nov. 2009), 26–37.

[9] F.J. Brandenburg. 1988. On the Intersection of Stacks and Queues. *Theoretical Computer Science* 58 (1988), 69–80.

[10] L. Breveglieri, A. Cherubini, and S. Crespi-Reghizzi. 1991. Real-time scheduling by queue automata. In *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFTS) (LNCS)*, J. Vytopil (Ed.), Vol. 571. Springer, Nijmegen, The Netherlands, 131–147.

[11] R. Buehrer and K. Ekanadham. 1987. Incorporating Dataflow Ideas into von Neumann Processors for Parallel Execution. *IEEE Transactions on Computers (T-C)* 36, 12 (December 1987), 1515–1522.

[12] D. Burger, S.W. Keckler, K.S. McKinley, M. Dahlin, L.K. John, C. Lin, C.R. Moore, J. Burrill, R.G. McDonald, and W. Yoder. 2004. Scaling to the End of Silicon with EDGE Architectures. *IEEE Computer* 37, 7 (July 2004), 44–55.

[13] A. Cherubini, C. Citrini, S. Crespi Reghizzi, and D. Mandrioli. 1991. QRT FIFO automata, breadth-first grammars and their relations. *Theoretical Computer Science* 85 (1991), 171–203.

[14] H. Corporaal. 1994. Design of Transport Triggered Architectures. In *Great Lakes Symposium on VLSI (GLSVLSI)*. IEEE Computer Society, Notre Dame, IN, USA, 130–135.

[15] H. Corporaal. 1999. TTAs: Missing the ILP complexity wall. *Journal of Systems Architecture* 45, 12-13 (June 1999), 949–973.

[16] H. Corporaal, J. Janssen, and M. Arnold. 2000. Computation in the Context of Transport Triggered Architectures. *International Journal of Parallel Programming* 28, 4 (August 2000), 401–427.

[17] M. Dahlem, A. Bhagyanath, and K. Schneider. 2018. Optimal Scheduling for Exposed Datapath Architectures with Buffered Processing Units by ASP. *Theory and Practice of Logic Programming (TPLP)* 18, 1 (January 2018), 438–451.

[18] G. Di Battista, P. Eades, R. Tamassia, and I.G. Tollis. 1994. Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry* 4 (1994), 235–282.

[19] G. Di Battista, F. Frati, and J. Pach. 2010. On the Queue Number of Planar Graphs. In *Foundations of Computer Science (FOCS)*. IEEE Computer Society, Las Vegas, NV, USA, 365–374.

[20] G. Di Battista, F. Frati, and J. Pach. 2013. On the Queue Number of Planar Graphs. *SIAM J. Comput.* 42, 6 (2013), 2243–2285.

[21] V. Dujmovic, G. Joret, P. Micek, P. Morin, T. Ueckerdt, and D.R. Wood. 2019. Planar Graphs have Bounded Queue-Number. In *Foundations of Computer Science (FOCS)*. IEEE Computer Society, Baltimore, MD, USA, 862–875.

[22] V. Dujmovic, G. Joret, P. Micek, P. Morin, T. Ueckerdt, and D.R. Wood. 2020. Planar Graphs Have Bounded Queue-Number. *Journal of the ACM (JACM)* 67, 4 (2020), 22:1–22:38.

[23] V. Dujmović. 2013. *Graph Layouts via Layered Separators.* Technical Report arXiv:1302.0304. arXiv.org.

[24] V. Dujmović, D. Eppstein, R. Hickingbotham, P. Morin, and D.R. Wood. 2021. *Stack-number is not bounded by queue-number.* Technical Report arXiv:2011.04195v2. arXiv.org.

[25] V. Dujmović, G. Joret, P. Micek, P. Morin, T. Ueckerdt, and D.R. Wood. 2020. *Planar graphs have bounded queue-number.* Technical Report arXiv:1904.04791v5. arXiv.org.

[26] V. Dujmović, P. Morin, and D.R. Wood. 2013. Layered Separators for Queue Layouts, 3D Graph Drawing and Nonrepetitive Coloring. In *Foundations of Computer Science (FOCS)*. IEEE Computer Society, Berkeley, CA, USA, 280–289.

[27] V. Dujmović, P. Morin, and D.R. Wood. 2019. *Queue Layouts of Graphs with Bounded Degree and Bounded Genus.* Technical Report arXiv:1901.05594v2. arXiv.org.

[28] P. Eades and D. Kelly. 1986. Heuristics for reducing crossings in 2-layered networks. *Ars Combinatoria* 21-A (1986), 89–98.

[29] P. Eades, B.D. McKay, and N.C. Wormald. 1986. On an Edge Crossing Problem. In *Australian Computer Science Conference.* Australian National University, 327–334.

[30] P. Eades and S. Whitesides. 1994. Drawing graphs in two layers. *Theoretical Computer Science (TCS)* 131, 2 (September 1994), 361–374.

[31] P. Eades and N.C. Wormald. 1994. Edge Crossings in Drawings of Bipartite Graphs. *Algorithmica* 11 (1994), 379–403.

[32] S. El-Basil. 1987. Applications of Caterpillar Trees in Chemistry and Physics. *Journal of Mathematical Chemistry* 1, 2 (1987), 153–174.

[33] J. Esparza, M. Luttenberger, and M. Schlund. 2014. A Brief History of Strahler Numbers. In *Language and Automata Theory and Applications (LATA) (LNCS)*, A.-H. Dediu, C. Martin-Vide, J.L. Sierra-Rodriguez, and B. Truthe (Eds.), Vol. 8370. Springer, Madrid, Spain, 1–13.

[34] M. Feller and M.D. Ercegovac. 1981. Queue machines: An organization for parallel computation. In *Conpar 81 (LNCS)*, W. Brauer, P. Brinch Hansen, D. Gries, C. Moler, G. Seegmüller, J. Stoer, N. Wirth, and W. Händler (Eds.), Vol. 111. Springer, Nürnberg, Germany, 37–47.

[35] U. Fößmeier and M. Kaufmann. 1997. Nice drawings for planar bipartite graphs. In *Italian Conference on Algorithms and Complexity (CIAC) (LNCS)*, G. Bongiovanni, D.P. Bovet, and G. Di Battista (Eds.), Vol. 1203. Springer, Rome, Italy, 122–134.

[36] G. Gange, P.J. Stuckey, and K. Marriott. 2011. Optimal k-Level Planarization and Crossing Minimization. In *Graph Drawing (GD) (LNCS)*, U. Brandes and S. Cornelsen (Eds.), Vol. 6502. Springer, Konstanz, Germany, 238–249.

[37] M.R. Garey and D. S. Johnson. 1983. Crossing Number is NP-Complete. *SIAM Journal on Algebraic Discrete Methods* 4, 3 (1983), 312–316.

[38] S. Gatzka and C. Hochberger. 2005. The AMIDAR Class of Reconfigurable Processors. *The Journal of Supercomputing* 32, 2 (2005), 163–181.

[39] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim. 2012. DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing. *IEEE Micro* 33, 5 (2012).

[40] F. Harary and A.J. Schwenk. 1973. The Number of Caterpillars. *Discrete Mathematics* 6 (1973), 359–365.

[41] P. Healy and A. Kuusik. 1999. The vertex-exchange graph: A new concept for multi-level crossing minimisation. In *Graph Drawing (GD) (LNCS)*, J. Kratochvíl (Ed.), Vol. 1731. Springer, Štiřín Castle, Czech Republic, 205–216.

[42] L.S. Heath, F.T Leighton, and A.L. Rosenberg. 1992. Comparing Queues and Stacks as Mechanisms for Laying out Graphs. *SIAM Journal on Discrete Mathematics* 5, 3 (August 1992), 398–412.

[43] L.S. Heath and S.V. Pemmaraju. 1996. Recognizing Leveled-Planar DAGs in Linear Time. In *Graph Drawing (GD) (LNCS)*, F.J. Brandenburg (Ed.), Vol. 1027. Springer, Passau, Germany, 300–311.

[44] L.S. Heath and S.V. Pemmaraju. 1997. Stack and Queue Layouts of Posets. *SIAM J. Comput.* 10, 4 (1997), 599–625.

[45] L.S. Heath and S.V. Pemmaraju. 1999. Stack and Queue Layouts of Directed Acyclic Graphs: Part II. *SIAM J. Comput.* 28, 5 (1999), 1588–1626.

[46] L.S. Heath, S.V. Pemmaraju, and A.N. Trenk. 1999. Stack and Queue Layouts of Directed Acyclic Graphs: Part I. *SIAM J. Comput.* 28, 4 (1999), 1510–1539.

[47] L.S. Heath and A.L. Rosenberg. 1992. Comparing Queues and Stacks As Machines for Laying Out Graphs. *SIAM J. Comput.* 21, 5 (October 1992), 927–958.

[48] J. Hoogerbrugge and H. Corporaal. 1994. Transport-Triggering vs. Operation-Triggering. In *Compiler Construction (CC) (LNCS)*, P. Fritzson (Ed.), Vol. 786. Springer, Edinburgh, UK, 435–449.

[49] R.A. Iannucci. 1988. Towards a Dataflow/von Neumann Hybrid Architecture. In *Int. Symposium on Computer Architecture (ISCA)*, H. Siegel (Ed.). IEEE Computer Society, Honolulu, Hawaii, USA, 131–140.

[50] T. Jain. 2019. *Nonblocking On-Chip Interconnection Networks.* Ph.D. Dissertation. Department of Computer Science, University of Kaiserslautern, Germany. PhD.

[51] D.S. Johnson. 1982. The NP-Completeness Column: An Ongoing Guide. *Journal of Algorithms* 3, 1 (1982), 89–99.

[52] P. Jääskeläinen, A. Tervo, G.P. Vayá, T. Viitanen, N. Behmann, and H. Blume. 2018. Transport-Triggered Soft Cores. In *International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE Computer Society, Vancouver, BC, Canada.

[53] M. Jünger, S. Leipert, and P. Mutzel. 1998. Level Planarity Testing in Linear Time. In *Graph Drawing (GD) (LNCS)*, S.H. Whitesides (Ed.), Vol. 1547. Springer, Montréal, Canada, 224–237.

[54] M. Jünger, S. Leipert, and P. Mutzel. 1998. A note on computing a maximal planar subgraph using PQ-trees. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 17, 7 (July 1998), 609–612.

[55] M. Jünger and P. Mutzel. 1996. Exact and heuristic algorithms for 2-layer straightline crossing minimization. In *Graph Drawing (GD) (LNCS)*, F.J. Brandenburg (Ed.), Vol. 1027. Springer, Passau, Germany, 337–348.

[56] M. Jünger and P. Mutzel. 1997. 2-Layer Straightline Crossing Minimization: Performance of Exact and Heuristic Algorithms. *Journal of Graph Algorithms and Applications* 1, 1 (1997), 1–25.

[57] G. Kahn. 1974. The Semantics of a Simple Language for Parallel Programming. In *Information Processing*, J.L. Rosenfeld (Ed.). North-Holland, Stockholm, Sweden, 471–475.

[58] G. Kahn and D.B. MacQueen. 1977. Coroutines and networks of parallel processes. In *Information Processing*, B. Gilchrist (Ed.). North-Holland, Toronto, Canada, 993–998.

[59] M. Kutrib, A. Malcher, C. Mereghetti, B. Palano, and M. Wendlandt. 2015. Deterministic input-driven queue automata: Finite turns, decidability, and closure properties. *Theo. Comp. Science* 578 (2015), 58–71.

[60] E.A. Lee and T. Parks. 1995. Dataflow Process Networks. *Proc. IEEE* 83, 5 (May 1995), 773–801.

[61] P. Mutzel. 1997. An alternative method to crossing minimization on hierarchical graphs. In *Graph Drawing (GD) (LNCS)*, S. North (Ed.), Vol. 1190. Springer, Berkeley, California, USA, 318–333.

[62] P. Mutzel and R. Weiskircher. 1998. Two-Layer Planarization in Graph Drawing. In *International Symposium on Algorithms and Computation (ISAAC) (LNCS)*, K.-Y. Chwa and O.H. Ibarra (Eds.), Vol. 1533. Springer, Taejon, Korea, 69–79.

[63] S.V. Pemmaraju. 1992. *Exploring the powers of stacks and queues via graph layouts.* Ph.D. Dissertation. Virigina Polytechnic Institute and State University, Blacksburg, VA, USA. PhD.

[64] B.R. Preiss. 1987. *Data Flow on a Queue Machine.* Ph.D. Dissertation. Department of Electrical Engineering, University of Toronto, Toronto, Ontario, Canada. PhD.

[65] B.R. Preiss and V.C. Hamacher. 1985. Data flow on a queue machine. In *International Symposium on Computer architecture (ISCA)*. IEEE Computer Society, Boston, MA, USA, 342–351.

[66] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, N. Ranganathan, D. Burger, S.W. Keckler, R.G. Mcdonald, and C.R. Moore. 2004. TRIPS: A Polymorphous Architecture for Exploiting ILP, TLP, and DLP. *ACM Transactions on Architecture and Code Optimization (TACO)* 1, 1 (2004).

[67] H. Schmit, B. Levine, and B. Ylvisaker. 2002. Queue machines: hardware compilation in hardware. In *Field-Programmable Custom Computing Machines (FCCM)*, J. Arnold and K.L. Pocek (Eds.). IEEE Computer Society, Napa, California, USA, 152–160.

[68] K. Schneider. 2021. Translating Structured Sequential Programs to Dataflow Graphs. In *Formal Methods and Models for Codesign (MEMOCODE)*, I. Saha and L. Zhang (Eds.). ACM, Beijing, China.

[69] K. Schneider, A. Bhagyanath, and J. Roob. 2022. Virtual Buffers for Exposed Datapath Architectures. In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV) (ITG-Fachbericht)*, J. Brandt (Ed.), Vol. 302. VDE, Virtual Event, 45–55.

[70] R. Sethi and J.D. Ullman. 1970. The Generation of Optimal Code for Arithmetic Expressions. *Journal of the ACM (JACM)* 17, 4 (October 1970), 715–728.

[71] K. Sugiyama, S. Tagawa, and M. Toda. 1981. Methods for Visual Understanding of Hierarchical System Structures. *IEEE Transactions on Systems, Man, and Cybernetics* SMC-l1, 2 (February 1981), 109–125.

[72] S. Swanson. 2006. *The WaveScalar Architecture.* Ph.D. Dissertation. University of Washington. PhD.

[73] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. 2003. WaveScalar. In *Microarchitecture (MICRO)*. IEEE Computer Society, San Diego, California, USA, 291–302.

[74] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, and S.J. Eggers. 2007. The WaveScalar Architecture. *ACM Trans. on Comp. Systems (TOCS)* 25, 2 (May 2007), 1–54.

[75] M.B. Taylor. 1999. *Design Decisions in the Implementation of a RAW Architecture Workstation.* Master's thesis. Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, USA. Master.

[76] M.B. Taylor, J.S. Kim, J.E. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffmann, P. Johnson, J.W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M.I. Frank, S.P. Amarasinghe, and A. Agarwal. 2002. The RAW Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. *IEEE Micro* 22, 2 (March/April 2002), 25–35.

[77] R. Vollmar. 1970. Über einen Automaten mit Pufferspeicherung. *Computing* 5, 1 (1970), 57–70.

[78] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, J. Babb, S. Amarasinghe, and A. Agarwal. 1997. Baring it all to Software: RAW Machines. *IEEE Computer* 30, 9 (September 1997), 86–93.

[79] V. Wiechert. 2017. On the Queue-Number of Graphs with Bounded Tree-Width. *The Electronic Journal of Combinatorics* 24, 1 (2017).

[80] F. Yazdanpanah, C. Alvarez-Martinez, D. Jimenez-Gonzalez, and Y. Etsion. 2014. Hybrid Dataflow/von-Neumann Architectures. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (June 2014), 1489–1509.