Rheinland-Pfälzische
Technische Universität
Kaiserslautern
Landau

Extending Allocations of Processing Units for Higher Instruction Level Paralleslism

Bachelor's Thesis

by

Ryan Stanley Antipow

June 1, 2025

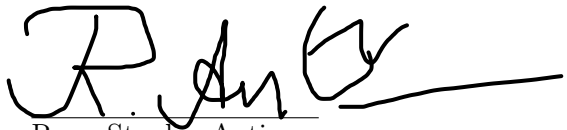
University of Kaiserslautern-Landau
Department of Computer Science
67663 Kaiserslautern
Germany

Examiner: Prof. Dr. Klaus Schneider
M.Sc. Nadine Kercher

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die von mir vorgelegte Arbeit mit dem Thema „Extending Allocations of Processing Units for Higher Instruction Level Paralleslism“ selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit — einschließlich Tabellen und Abbildungen —, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Kaiserslautern, den 1.6.2025



Ryan Stanley Antipow

Abstract

This thesis solves the problem of improving instruction-level parallelism (ILP) in dataflow-based processor architectures by optimizing scheduling and processing unit (PU) allocation. In particular, it addresses the Buffered Exposed Datapath (BED) processor SCAD (Synchronous-Control Asynchronous-Dataflow), which was designed to maximize parallel execution, avoiding conventional memory bottlenecks and exposing internal structures directly to the compiler. Based on the Averest framework, the thesis designs and tests a scheduling algorithm that transforms sequential MiniC programs into Dataflow Process Networks (DPNs), assigns operations to Processing Units (PUs), and produces executable move code while increasing initial PU allocations to facilitate faster execution.

The key contribution is a post-SAT scheduling algorithm that leverages a previously calculated minimal PU allocation to utilize more processing units, thereby enhancing parallelism without introducing new data dependencies or critical crossings. A novel mechanism, referred to as trace juggling, is introduced to balance task distribution and reduce execution time by interleaving operations from different traces while preserving strict First-In-First-Out (FIFO) semantics.

Experimental tests on representative benchmarks, such as MinTwoPUsNeeded and Rainbow DPNs, demonstrate the effectiveness of the extended allocation strategy. The scalability and efficiency of the approach are further verified by results that demonstrate a significant reduction in execution cycles - from 18 to 12 cycles in one example - by adding just a few PUs. However, critical failure points are also identified in generated schedules due to the mishandling of duplication nodes with switched outputs. These results suggest that DPN models and scheduler implementations should be refined for broader application areas.

Finally, this thesis presents a practical implementation and a conceptual improvement of PU scheduling for exposed datapath architectures. It provides the foundation for further development of resource-aware scheduling algorithms, efficient duplication semantics treatment, and incorporation into hardware design flows for high-performance parallel systems.

Zusammenfassung

Diese Arbeit befasst sich mit der Verbesserung der Instruction-Level-Parallelität (ILP) in datenflussbasierten Prozessorarchitekturen durch Optimierung des Schedulings und der Zuordnung von Verarbeitungseinheiten (PUs). Der Fokus liegt insbesondere auf dem Buffered Exposed Datapath (BED)-Prozessor SCAD (Synchronous-Control Asynchronous-Dataflow), der für maximale parallele Ausführung entwickelt wurde, um konventionelle Speicherengpässe zu vermeiden und interne Strukturen direkt dem Compiler zugänglich zu machen. Basierend auf dem Averest-Framework entwickelt und testet die Arbeit einen Scheduling-Algorithmus, der sequentielle MiniC-Programme in Dataflow Process Networks (DPNs) transformiert, Operationen Verarbeitungseinheiten (PUs) zuweist und ausführbaren Move-Code erzeugt. Gleichzeitig werden die anfänglichen PU-Zuweisungen erhöht, um eine schnellere Ausführung zu ermöglichen.
Der wichtigste Beitrag ist ein Post-SAT-Scheduling-Algorithmus, der eine zuvor berechnete minimale PU-Zuweisung verwendet, um mehr Verarbeitungseinheiten zu nutzen und so die Parallelität zu verbessern, ohne neue Datenabhängigkeiten oder kritische Kreuzungen einzuführen. Ein neuartiger Mechanismus, das sogenannte Trace-Juggling, soll die Aufgabenverteilung ausbalancieren und die Ausführungszeit reduzieren. Dazu werden Operationen verschiedener Traces verschachtelt, wobei die strikte First-In-First-Out-Semantik (FIFO) gewahrt bleibt.
Experimentelle Tests mit repräsentativen Benchmarks, wie MinTwoPUsNeeded und Rainbow DPNs, belegen die Wirksamkeit der erweiterten Allokationsstrategie. Die Skalierbarkeit und Effizienz des Ansatzes werden durch Ergebnisse bestätigt, die eine signifikante Reduzierung der Ausführungszyklen – in einem Beispiel von 18 auf 12 Zyklen – durch das Hinzufügen nur weniger PUs belegen. Jedoch wurden auch kritische Fehlerpunkte in generierten Schedules identifiziert, die auf die fehlerhafte Handhabung von Duplikationsknoten mit getauschten Ausgängen zurückzuführen sind. Diese Ergebnisse legen nahe, dass DPN-Modelle und Scheduler-Implementierungen für breitere Anwendungsbereiche weiterentwickelt werden sollten.

Abschließend präsentiert diese Arbeit eine praktische Implementierung und eine konzeptionelle Verbesserung des PU-Scheduling für exponierte Datenpfadarchitekturen. Es bietet die Grundlage für die Weiterentwicklung ressourcenbewusster Planungsalgorithmen, eine effiziente Behandlung der Duplizierungssemantik und die Einbindung in Hardware-Designabläufe für leistungsstarke parallele Systeme.

Contents

1 Introduction

Processor architecture is essential to everyday life as it defines how a computer analyzes data and executes instructions. It directly impacts computer systems' performance, efficiency, and scalability. A well-designed processor allows for quicker data processing, lower power consumption, and more multitasking, all required for current applications ranging from mobile devices to supercomputers. Furthermore, architectural decisions influence software development since programs must be tailored to the processor's specific instruction sets and features. Different architectures are tailored for specific applications, ranging from embedded systems to high-performance computing, affecting everything from performance to energy efficiency. Overall, processor architecture shapes the capabilities and future of computing technologies.

A widely known metric for the evolution of processors is Moore's Law, developed by Gordon Moore in 1965, which states that the number of transistors on a microchip doubles roughly every two years, resulting in an exponential rise in computer capacity [Moo65]. Nevertheless, this rule is limited because transistors are becoming almost atomic in size while costs remain constant [Lei+20]. These constraints necessitate advancements in processor architecture that go beyond merely adding more transistors. In order to optimize task allocation and lessen dependency on shared memory, current trends point to a shift towards architectures that expose internal components to the compiler. This change minimizes and, if possible, eliminates the requirement for shared memory. It overcomes the drawbacks of memory structures that cannot keep up with technological advancements by giving compilers control over scheduling, data transfers, dependency analysis, and processing unit allocation. This change has led to the emergence of several exposed datapath architectures, such as RAW/Tilera [Aga99; Tay99; Tay+02; Wai+97; Wai04], TRIPS [Bur+04; San+03; San+04], DySER [Gov+12], AMIDAR [GH05], TTA [AC97; HC94a; Hoo97; HC92; HC94b; Jää+18; CL95; Cor99; CJA00; Cor94; KC97], and SCAD [ABS18; Bha21; BS16; SBR22b; Ker23].

The Synchronous Control and Asynchronous Dataflow (SCAD) processor, as aforementioned, is an exposed datapath processor developed by the Rheinland-Pfälzische Technische Universität Kaiserslautern-Landau. The SCAD processor improves processing efficiency by asynchronous data transfer, allowing many processes to run concurrently without waiting for one to finish before the next [SBR22b]. Using FIFO (First In, First Out) buffers optimizes asynchronous execution even further. These buffers serve as temporary storage locations for data and instructions, ensuring that activities are executed in the

order they were received. The FIFO structure is necessary because it maintains the integrity and order of execution operations, preventing data from being processed out of sequence [SBR22b]. As a consequence, the SCAD processor not only improves processing performance by handling many processes concurrently, but it also maintains a systematic approach to memory access.

This thesis introduces a scheduling algorithm that utilizes instruction layer parallelism as efficiently as possible while extending existing allocations. This algorithm is specifically designed to generate schedules for the SCAD processor. A detailed description of the SCAD processor is provided in Chapter 2.3.

1.1 General Problem Setting

BED processors are a subset of exposed datapath architectures that reveal their internal structure to compilers, allowing compilers to see the hardware capabilities and layout of the processor directly and optimize the code more efficiently when compiling. The available resources and internal structure must be considered when generating code for a BED architecture. It is essential to identify which processing unit should carry out which specific action, which data moves through the processor, and how local memory should be utilized. Therefore, scheduling algorithms play a crucial role in optimizing the allocation of resources and managing tasks. These algorithms determine the order in which jobs are executed, ensuring efficient processing while minimizing delays and maximizing resource utilization. From real-time systems that require immediate task execution to batch processing, where tasks can be handled at predetermined times, scheduling algorithms vary widely in their implementation and objectives. By effectively organizing tasks, scheduling algorithms enhance overall system performance, ensuring that resources are used efficiently and objectives are met. Most scheduling algorithms, including the one implemented and described in Chapter 4 and the scheduling algorithms presented in 3, aim to achieve the fastest computational time possible with the given resources. This is achieved through the parallel execution of nodes on different processing units, if possible. Otherwise, a sequential execution should be considered. When their input data is available, these processing units execute and can therefore operate independently.

The starting point of this thesis will be a predefined program provided in the MiniC language, a language developed for teaching processor architectures [BSS14]. Afterward, a method for translating structured, sequential programs into equivalent dataflow process networks (DPNs) using a designated set of nodes for token control, control flow, memory access, and data operations is applied to visualize the problem and as an intermediate representation for the compilation of the DPN [Sch21]. A more detailed review of DPNs will be provided in Chapter 2.2.

1.2 New Contributions

This thesis contributes a scheduling algorithm based on a generated dataflow process network and a minimal allocation of processing units resulting from a MiniC program. This scheduling algorithm utilizes instruction layer parallelism for the fastest possible program execution. The Averest framework is utilized for this translation. Averest includes a collection of tools designed for specifying, verifying, and implementing reactive systems [SS05]. The scheduling algorithm allocates the given SCAD processor with a predefined number of PU and automatically generates the needed SCAD code to run the program. Additionally, the provided scheduling algorithm should translate any MiniC program into an equivalent SCAD program, utilizing resources optimally. However, as shown in chapter 5.4, critical failure points are also identified in generated schedules due to the mishandling of duplication nodes with switched outputs. Therefore, further refinement is needed to consider this aspect.

1.3 Outline of the Thesis

In Chapter 2 establishes the background by introducing key concepts. It presents MiniC, a low-level programming language for writing programs that are to be converted to Dataflow Process Networks (DPNs). The concept and structure of DPNs are described, and the conversion of sequential MiniC programs into this intermediate form is described. The architecture of the SCAD processor is described below, together with its asynchronous data flow, FIFO buffer mechanics, and components (processing units, load/store units, and control logic). Instruction-level parallelism is then introduced in the context of buffered exposed datapath (BED) architectures. This is followed by a mapping of dataflow graphs to DPN architectures with SAT-based constraints for processing unit allocation and scheduling consistency. The chapter ends with a formal definition of the logical constraints that lead to FIFO behavior and avoid critical scheduling conflicts.
Chapter 3 describes describes various scheduling methods, ranging from the simple ASAP (As Soon As Possible) to the ALAP (As Late As Possible) algorithm. It then investigates trace scheduling, a more advanced technique for optimizing instruction execution over control flow traces. Vertex-disjoint paths are proposed as a method to maximize ILP by dividing task chains into independent processors.Efficiency metrics for evaluating scheduling algorithms, how well they optimize runtime, resource use, and parallelism, with less emphasis on scheduling time in static systems, are discussed. The chapter concludes with an examination of the optimal schedule for implementing DPNs on SCAD processors under unconstrained resource scenarios, followed by the consideration of limited constraints. A technique named Trace Juggling is proposed to enable the execution of operations across traces, thereby maximizing the utilization of available resources and avoiding execution bottlenecks.
Chapter 4 presents the implementation of the proposed scheduling algorithm

based on the Averest framework. It describes how current minimal allocations are extended under some constraints and presents the system architecture for storing experimental data and results. The chapter describes the steps involved in parsing MiniC code, translating it into DPNs, obtaining SAT-based allocation constraints, and preparing the graphs for scheduling. It also describes the implementation of the scheduling algorithm, visualization of results and optimization of runtime efficiency.
In Chapter 5, the scheduling algorithm is experimentally tested. It outlines the experimental setup and provides a detailed analysis of one specific case, followed by broader tests across various benchmarks. In Section 5.4 the failures of the algorithm is also discussed, primarily due to incorrect scheduling resulting from mishandling of duplicated nodes with swapped outputs.
It is concluded that its contributions were mainly the design and implementation of an extended PU allocation and scheduling algorithm that enhances ILP for SCAD processors. It focuses on the practical benefits illustrated by performance gains in the limited benchmark tests.
Finally, the thesis discusses potential directions for future work. These include reworking the handling of duplication semantics , scheduling robustness, and integrating with hardware design flows for use cases in high-performance parallel computing.

2 Fundamentals

2.1 MiniC

MiniC is a programming language devoloped by the Rheinland-Pfälzische Technische Universität Kaiserslautern-Landau for minimal multi-threaded applications. Its development focuses on a compact set of data types, which include booleans, unsigned/signed integers, arrays and tuples [BSS14]. Booleans represent true and false values, enabling logical operations and condition checks within the program. Unsigned integers, represented as 'nat' (natural numbers) and signed integers, represented as 'int', allowing for both positive and negative values. Arrays enable the use of collections of elements that can be accessed via indices. Tuples allow groups of data types to be bundled together, enabling more complex data representations.

A typical MiniC program is structured into three sections organizing the code. The sections are defined as follows:

- **global/shared variables**, dedicated to variables accessible from any function or thread within the program. The design necessitates careful management of these variables, as they must conform to a weak memory model [BSS14].

- **Function Declarations** are callable from any active thread or function within the program, but these functions can not be recursive to avoid runtime stacks [BSS14].

- **Threads** are each statically allocated to a core, which means that threads are treated as independent main programs capable of executing concurrently on different cores. The synchronization of these threads is crucial, as it allows for cooperative multitasking within the program, optimizing performance by effectively leveraging multi-core architectures [BSS14].

For further reference, see MiniC reference card[1] provided by the Embedded Systems Group of the University of Kaiserslautern. This resource includes essential commands and syntax rules aiding in the utilization of the MiniC programming environment.

[1] MiniC reference card: https://es.cs.rptu.de/tools/teaching/MiniCRef.pdf

2.2 Dataflow Process Networks

2.2.1 The concept of Dataflow Process Networks

According to the determinations in [Sch21], the following discussion explores Dataflow Process Networks (DPN), a computational model used to represent the data flow in a system. It consists of various process nodes and directed edges combined into a directed graph where each edge represents a communication channel. In this structure, nodes are responsible for processing data. A node can execute its operation, also called the node can fire, only when certain conditions are met, more precisely if a sufficient number of tokens arrive in its input buffers. The incoming edges of the corresponding node define the number of tokens needed to fire. When these criteria are satisfied, the node extracts the necessary input values from the front of its input buffers and executes its designated calculation or processing operation. The node generates output values after it has completed its operation. The results are placed at the end of its output buffers, ready for use by other connected nodes in the graph.

Because of the nature of dataflow networks, processing can occur in a highly parallel form. As long as the required data is available, several nodes can fire simultaneously. Dataflow graphs are convenient for modeling complex data processing tasks, where the order of operations is dictated by data availability rather than a pre-established command sequence. An exemplary DPN visualization is provided in Figure 2.2 for further illustration of this concept.

2.2.2 From Sequential Programs to Dataflow Process Networks

The translation from Sequential Programs to Dataflow Process Networks is defined recursively across the statements and will generate a DPN corresponding to the statement. The following nodes, as shown in Figure 2.1, can be used to transform the program into a DPN. Nodes take in and generate values, known as tokens; tokens used are read from the front of the input buffers and then removed, while new tokens are added to the end of the output buffers. This is realized with directed edges between the Nodes. Each node has a specified number of tokens needed for its input buffers, though each buffer may have different requirements. When the necessary tokens are present, the node can fire, meaning it will consume the tokens, perform its computations, and generate the resulting tokens.

2.2.3 Graphical representation of Dataflow Process Networks

In the graphical representation, various edge styles, which will be explained shortly, serve to illustrate the connections between outputs and inputs that the buffer manages. These distinct styles visually convey the nature of these relationships. This information is effectively represented in the DPN shown in Figure 2.2, which visually translates the example program called DAXPY. DAXPY (Double-precision A $\cdot$ X Plus Y) is a basic linear algebra operation

C is a copy node with one input and one output buffer. It consumes the head of its input and outputs it to the output buffer, useful for leveling in a DPN.

D is a duplication node with one input and two output buffers, creating multiple instances of a input when it fires

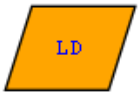
S is a swap node with two input and two output buffers; it swaps the heads of its input buffers and sends them to the opposite output buffers.

J is a join node with two input and one output buffer; it consumes one token from each input buffer and produces one token in its output, regardless of the values.

Kill is a kill node that destroys the head of its input buffer when it fires.

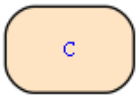
SEL (select) has three input buffers and forwards a value from either x_1 or x_0 to output y depending on the boolean in buffer c.

SWT (switch) takes two input buffers and two output buffers; it can forward a value from x to either y_1 or y_0 based on the boolean value in c.

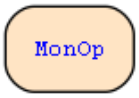


$\mathrm{LD_a}$ (load memory a) consumes tokens to read from memory at a specified address and outputs the loaded value.

$\mathrm{ST_a}$ (store memory a) takes inputs to store a value in memory at a specified address, enabling future memory access.



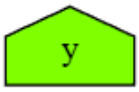
Const(c) always fires, outputting a constant value c.



MonOp(f, x) performs unary or binary operations on value from the input buffers.

BinOp($\odot$, x0, x1) performs binary operations on values from their input buffers.

x is a value that is recevied from the Data Memory



y is a value that is pushed to the Data Memory

Figure 2.1: *DPN nodes used in the translation, derived from [Sch21] and visualized using the Averest framework [SS05]*

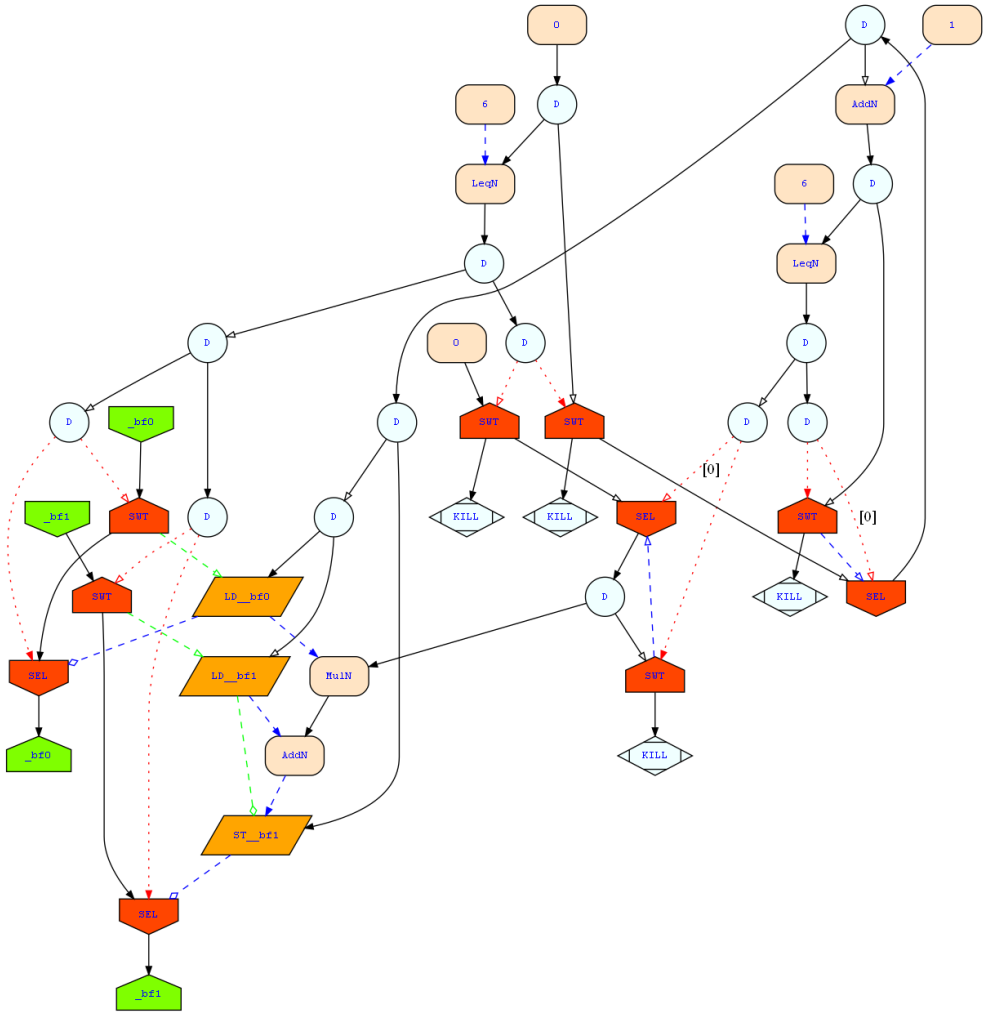


Figure 2.2: *A visualisation of the a DPN used to calculate DAXPY (Double precision $A \cdot \vec{X}$ Plus $\vec{Y}$), using the Averest framework [SS05]*

that computes the result a · X + Y, where a is a scalar and X and Y are vectors. The figure not only highlights the structural layout but also aids in comprehending how the components interact within the program's flow.

Once we have addressed the specific cases for the inputs, which include a condition input (SEL and SWT) marked in red and token inputs for LD and ST displayed in green, we can proceed to handle the remaining inputs with a standard approach. The token inputs represent the variables the program needs to execute. The output tokens resemble the results the program calculates while executing.

In this context, we represent the first (left) input with a solid black arrow. This visual cue helps to easily distinguish it from the second (right) input, which is denoted by a blue dashed arrow, following the established syntax for the node.

Similarly, for the outputs, we apply a consistent visual strategy. Token outputs are indicated by an arrowhead shaped like a diamond, clearly distinguishing them from other types of outputs. For non-token outputs, we use filled arrowheads to signify the first (left) output, while an empty arrowhead is used for the second (right) output. This layered visual representation assists in clarifying the flow and relationships of inputs and outputs within the system.

2.3 Synchronous-Control Asynchronous-Dataflow

The Synchronous-Control Asynchronous-Dataflow (SCAD) processor, developed by the Embedded Systems Group at the Rheinland-Pfälzische Technische Universität Kaiserslautern-Landau, is a Buffered Exposed Datapath (BED) processor. BED processors are a subcategory of exposed datapath architectures that make their internal structure accessible to the compiler. Therefore, compilers can directly see the processor's hardware capabilities and layout. This exposure allows for better optimization during compilation, enabling compilers to produce more efficient code.

The term synchronous control denotes the processor's capability to control operations in order, guaranteeing the correctness of the program's execution. Meanwhile, Asynchronous-Dataflow refers to the separate data processing from control signals, enabling higher and independent throughput.

In the BED architecture, the input and output ports of the processing units utilize First-in, First-out (FIFO) buffers. Its programming uses move instructions, which transfer values from the head of output buffers to the tail of input buffers.

Figure 2.3 shows a general template of a BED Architecture wich consists of Processing Cores, Load/Store Unit, Control Unit, Programm memory, Data Memory and a Interconnection Network.

2.3.1 Processing Core/ Processing Unit

A Processing Unit (PU) executes instructions received in the input buffers. Figure 2.4 shows that each PU has three input and two output buffers. The

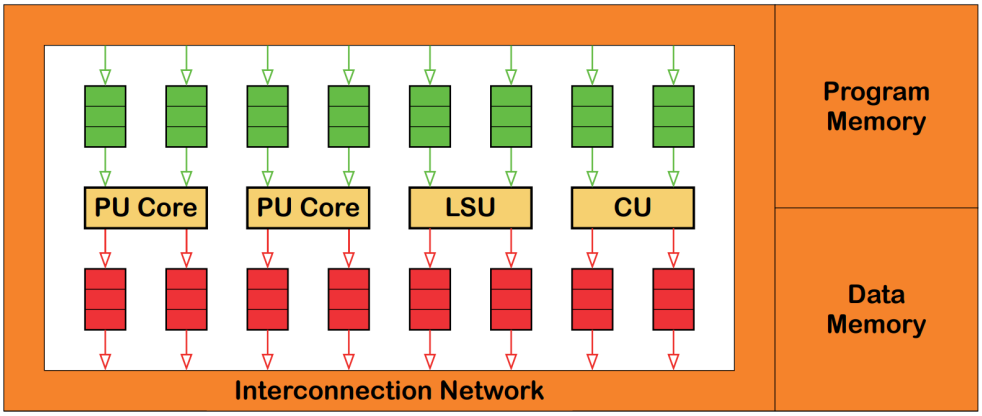


Figure 2.3: *"General Template of a BED Architecture"[SBR22b]*

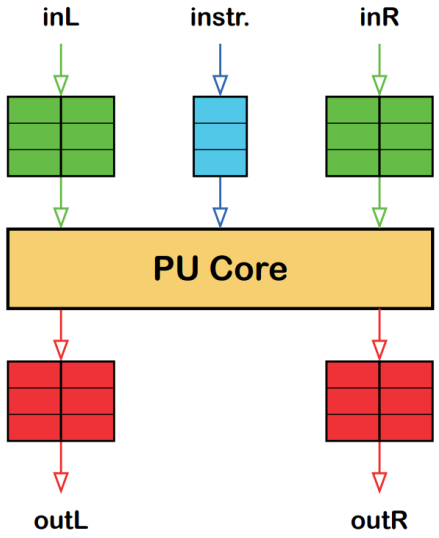


Figure 2.4: *"Processing Unit in a BED Architecture with Virtual FIFO Buffers"[SBR22b]*

instruction buffer highlighted in blue is optional and only required if the PU is capable of executing multiple instructions [Ker23]. Pairs of entries of the format (adr, val) are stored in Buffers inL, inR, outL, and outR. When referring to an input buffer, 'adr' denotes the address of the output buffer that is part of the PU that has produced or will produce the value 'val'. An '$\perp$' in val means the required value is unavailable and will be supplied later from the output buffer linked to 'adr'. Likewise, 'adr' in an output buffer denotes the address of the PU's input buffer configured to receive the value [BS17].

2.3.2 Load/Store Unit and Data Memory

The load/store unit (LSU) is responsible for reading from or writing to the data memory and has exclusive access to the data memory. As a result, it must

obtain an opcode to decide whether to load or store data. LSU needs two input buffers for storing, one for the memory addresses and another for the values to be stored at the corresponding addresses for saving data. Data loading needs just one input buffer for the addresses, and an output buffer value loaded from memory [BS17]. Utilizing a single LSU prevents weak memory from occurring [Ker23].

2.3.3 Control Unit and Programm memory

The control unit (CU) retrieves the subsequent move instruction (src, tgt), depending on the program counter, from the instruction memory and transmits it via the move-instruction bus (MIB), which is part of the Interconnection Network to all PUs. The input buffer at address tgt will append the entry (src, $\perp$) to its end, and the output buffer at address src will append the entry (tgt, $\perp$). A feedback signal indicating a full buffer will be sent to the control unit if a buffer fills up. Additionally, the other buffer will not save the entry, and the control unit will momentarily stop the subsequent cycle and repeat the move instruction. A move instruction's (src, tgt) related data transfer will be delayed. As a result, the control flow runs in tandem with the data flow, which happens asynchronously [BS17].

2.3.4 Interconnection Network

The interconnection network of the SCAD processor comprises two primary components: the move-instruction bus (MIB) and the data transport network (DTN). The move-instruction bus is used for the synchronous transmission of values from the control unit, while the data transport network enables asynchronous communication of values among components as they become available [BS17].

2.4 Instruction-Level Parallelism

The capacity of a processor to carry out many instructions simultaneously is known as Instruction-Level Parallelism (ILP). This technique improves throughput and performance using the inherent parallelism in instruction execution. Static scheduling, first created for lengthy instruction words, and dynamic scheduling for processors with out-of-order execution are two traditional approaches to using ILP in processor designs [BS17].

However, buffered exposed datapath designs may also be able to use the maximum ILP [BS17]. Future processor designs for high-performance computing must maximize ILP in exposed datapath architectures.

2.5 Mapping Dataflow Graphs to DPN architectures

According to the article "Consistency Constraints for Mapping Dataflow Graphs to Hybrid Dataflow/von Neumann Architectures" [SB23], allocating a Dataflow

Graph (DFG) to the BED architecture can be accomplished as described in the following.
Following the initial stage of converting a sequential program into a DFG, the DFG must be allocated to the BED architecture to generate the move code. Dataflow graphs are computational models where nodes represent operations and edges represent data paths, implemented as FIFO (First-In-First-Out) buffers.

A Dataflow Graph, or a DFG ($\mathcal{P}$, $\mathcal{B}$), comprises a set of nodes $\mathcal{P}$ interconnected by edges $\mathcal{B}$. The nodes $i \in \mathcal{P}$ realize one of the operations listed in Figure 2.1.
A DPN architecture refers to a DPN ($\mathcal{U}$, $\mathcal{C}$) comprising nodes $\mathcal{U}$ = {CU, IC, PU[0], ... , PU[n - 1]} and edges $\mathcal{C}$, which are also known as channels. CU, known as the control unit, transmits instructions to all other nodes. The interconnection network IC is responsible for transporting data between the PUs, and n PUs execute instructions.
In a given dataflow graph ($\mathcal{P}$, $\mathcal{B}$) and a DPN architecture ($\mathcal{U}$, $\mathcal{C}$), the synthesis problem involves the following tasks while adhering to the FIFO behavior of the buffers:

- Define a PU allocation α: $\mathcal{P} \rightarrow \mathcal{U}$ that assigns the nodes $\mathcal{P}$ of the dataflow graph to the processing units (PUs) within the DPN architecture.
- Establish a strict and total ordering $\prec$ of the nodes $\mathcal{P}$, outlining a schedule for executing the nodes on the PUs of $\mathcal{U}$.
- Create a strict and total ordering $\sqsubset$ of the edges $\mathcal{B}$, which specifies the data transfers needed to move values from the output buffers to the input buffers of the PUs in $\mathcal{U}$ via the interconnection network IC.

The mapping must respect FIFO behavior, ensuring that data is consumed in the same order it was produced. In [SB23], they define consistency rules to avoid conflicts when different data elements are assigned to the same source or target buffers.

In a dataflow graph ($\mathcal{P}$, $\mathcal{B}$), when mapped to a DPN architecture ($\mathcal{U}$, $\mathcal{C}$) with a given processing unit allocation α, a node execution order $\prec$, and an edge (or buffer) ordering $\sqsubset$, we can examine the interaction between two edges. Consider two edges $x_1 : p_1 \rightarrow q_1$ and $x_2 : p_2 \rightarrow q_2$, where $p_1 \prec p_2$ and $q_2 \prec q_1$. Under these conditions, assuming p_1 and p_2 write to a buffer and q_1 and q_2 read from a buffer

- Node p_1 must write to a buffer before node p_2, due to $p_1 \prec p_2$.
- Node q_2 must read data before node q_1, due to $q_2 \prec q_1$.

Now considering these edges, based on the PU allocation α and buffer assignments and generated move instructions src $\rightarrow$ tgt. The source address src

is an output buffer PU[i].outL or PU[i].outR and the destination address tgt, is one of the buffers PU[i].inL or PU[i].inR with i $\in \alpha$, this generates move instructions in the form of $src_1 \rightarrow tgt_1$ and $src_2 \rightarrow tgt_2$, and four possible cases arise.

1. $src_1 \neq src_2$, and $tgt_1 \neq tgt_2$, this indicates that $src_1 \rightarrow tgt_1$ and $src_2 \rightarrow tgt_2$ every order of the move instructions is possible.

2. $src_1 = src_2$, and $tgt_1 \neq tgt_2$, it is critical that $src_1 \rightarrow tgt_1$ occurs before $src_2 \rightarrow tgt_2$, due to $p_1 \prec p_2$.

3. $src_1 \neq src_2$, and $tgt_1 = tgt_2$, the requirement is that $src_2 \rightarrow tgt_2$ must be executed before $src_1 \rightarrow tgt_1$, due to $q_2 \prec q_1$.

4. $src_1 = src_2$, and $tgt_1 = tgt_2$, the requirement is that $src_1 \rightarrow tgt_1$ must occure before $src_2 \rightarrow tgt_2$, due to $p_1 \prec p_2$, but on the other hand, due to $q_2 \prec q_1$, $src_2 \rightarrow tgt_2$ must be executed before $src_1 \rightarrow tgt_1$. This indicates that there is no correct move code sequence.

Case four represents a critical crossing where two data tokens sharing the same buffer cannot be scheduled without violating FIFO behavior due to conflicting production and consumption orders, with a more detailed proof can be found in [SBR22a] . It represents a scheduling deadlock that must be resolved through architectural or compilation strategies. Identifying and eliminating critical crossings is essential to ensure correct, efficient, and parallel execution in dataflow-based systems like the SCAD processor.

A central challenge in this mapping is enforcing consistency constraints, ensuring that data tokens maintain FIFO behavior and that no critical crossings occur. To systematically capture and verify these constraints, we can translate the mapping problem into a Boolean satisfiability (SAT) problem. In particular, many consistency requirements, such as the relative ordering of node firings and token movements, can be expressed using simple binary (two-variable) logical clauses, making them suitable for encoding as a 2-SAT problem. A 2-SAT formulation allows us to:

- Represent orderings and PU assignments as binary decision variables

- Enforce FIFO-consistent execution using implications (e.g., "if node A precedes node B, then edge x must precede edge y")

- Efficiently determine whether a conflict-free, valid schedule exists using a known polynomial-time algorithm for 2-SAT

Thus, the complex task of scheduling and resource assignment in dataflow architectures can be reduced to solving a logically structured problem, enabling automated tools to explore the feasibility of the mapping and generate consistent schedules with minimal computational overhead.

2.6 Constraints for Dataflow Graphs

For any dataflow graph consisting of nodes $\mathcal{P}$ and buffers $\mathcal{B}$, along with any DPN architecture featuring ϱ PUs, Klaus Schneider et al. established in their paper [SBR22a] constraints through propositional variables $\alpha_{p,k}$, which indicate that node p is allocated to PU k. Additionally, they utilize a binary relation $\prec$ among the nodes that represents their scheduling, and a binary relation $\sqsubset$ concerning the buffers that indicates the scheduling of data transfers. To achieve this, they start by defining the following predicates for nodes p_1, p_2, and buffers b_1, b_2, the functions inBf(b) and outBf(b) specify which input or output port of a PU is utilized by buffer b. This now holds the following:

- $\mathsf{samePU}(p_i, p_j) :\Leftrightarrow \bigvee_{k=1}^{\varrho} \alpha_{p_i,k} \wedge \alpha_{p_j,k}$
- $\mathsf{srcEQ}(b_i, b_j) :\Leftrightarrow \mathsf{outBf}(b_i){=}\mathsf{outBf}(b_j) \wedge \mathsf{samePU}(p_i, p_j)$
- $\mathsf{tgtEQ}(b_i, b_j) :\Leftrightarrow \mathsf{inBf}(b_i){=}\mathsf{inBf}(b_j) \wedge \mathsf{samePU}(q_i, q_j)$

With these variables, relations, and predicates, they now define the following constraints to encode the synthesis problem for ϱ PUs:

- $\prec$ is a strict order on all nodes:
  - irreflexivity: $\neg(p \prec p)$ must hold for all nodes $p \in \mathcal{P}$
  - transitivity: for all nodes $p_1, p_2, p_3 \in \mathcal{P}$, they demand $p_1 \prec p_2 \wedge p_2 \prec p_3 \rightarrow p_1 \prec p_3$
  - totality: for all nodes $p_1, p_2 \in \mathcal{P}$ with $p_1 \neq p_2$, they demand $p_1 \prec p_2 \vee p_2 \prec p_1$
- $\sqsubset$ is a order on all buffers:
  - irreflexivity: $\neg(b \sqsubset b)$ must hold for all buffers $b \in \mathcal{B}$
  - transitivity: for all buffers $b_1, b_2, b_3 \in \mathcal{B}$, they demand the constraint $b_1 \sqsubset b_2 \wedge b_2 \sqsubset b_3 \rightarrow b_1 \sqsubset b_3$
  - totality: for all buffers $b_1, b_2 \in \mathcal{B}$ with $b_1 \neq b_2$, they demand $b_1 \sqsubset b_2 \vee b_2 \sqsubset b_1$
- data dependencies: for every buffer $b : p \rightarrow q$, they demand the node ordering constraint $p \prec q$
- PU allocation: every node $p \in \mathcal{P}$ is mapped to one PU

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

- FIFO behavior constraints: for all buffers $b_i : p_i \rightarrow q_i$ and $b_j : p_j \rightarrow q_j$, the following constraints ensure the consistency of the node ordering $\prec$ and the buffer ordering $\sqsubset$:
  - $p_i \prec p_j \wedge \mathsf{srcEQ}(b_i, b_j) \rightarrow b_i \sqsubset b_j$
  - $q_j \prec q_i \wedge \mathsf{tgtEQ}(b_i, b_j) \rightarrow b_j \sqsubset b_i$

3 Scheduling

Scheduling is the process of controlling tasks, instructions, or processes to maximize use of resources and performance. This involves identifying the sequential order of operations and effectively managing program resources. Several scheduling algorithms exist, each designed for specific environments and goals. Many variations and hybrids are also designed to handle specific needs in real-time, distributed, or multiprocessor systems. In the following sections, some scheduling algorithms, especially those significant for scheduling algorithms for the SCAD processor, will be presented and extended to create a new variation of a scheduling algorithm to maximize the use and performance of resources tailored to the specific constraints of the SCAD processor.

3.1 ALAP and ASAP Sheduling

Two simple scheduling techniques were described in the "Introduction to the Scheduling Problem" by Robert A. Walker and Samit Chaudhuri [WC95]. According to their description, the following two simple scheduling algorithms can be described as follows.
ASAP (As Soon As Possible) and ALAP (As Late As Possible) scheduling are the two basic algorithms utilized to solve the unconstrained scheduling (UCS) problem. The UCS problem describes assigning operations while respecting all data dependencies and assuming unlimited computational resources. ASAP scheduling places each operation into the earliest control step available, scheduling them one at a time. A control step represents a discrete unit of time during which specific operations or instructions are executed. One control step can include multiple instructions that can be executed in parallel. ALAP scheduling operates similarly but allocates each operation to the latest possible control step. Adding resource constraints (RCS) and time constraints (TCS) also extends the UCS to consider limited hardware resources and a fixed number of time steps, respectively.

To illustrate the concepts of ASAP and ALAP scheduling more concretely, let's consider a simple example involving a set of additions that must be planned while respecting their dependencies. Imagine we have the following calculation y = (a+b) + (c+d) + (e+f), which is based on the example in [WC95] and illustrated as a DPN in Figure 3.1. ASAP scheduling algorithm will schedule the addition (a+b), (c+d), and (e+f) in control step 1 as depicted in Figure 3.2. In contrast, the ALAP scheduling algorithm will only schedule the addition (a+b) and (c+d) in control step 1 and push (e+f) to control step 2, as shown in Figure 3.3.

While ASAP and ALAP are greedy algorithms and provide quick resolu-

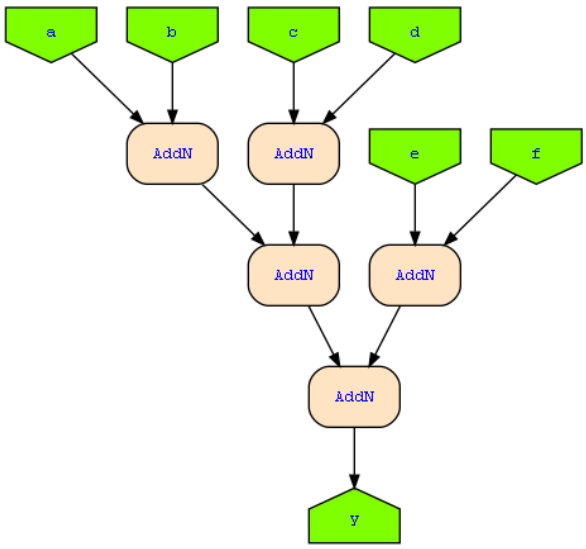


Figure 3.1: *A simple addition y = (a+b) + (c+d) + (e+f), based on the example presented in [WC95]*

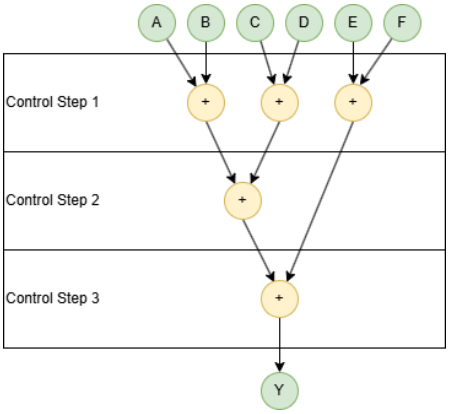


Figure 3.2: *ASAP schedule for 3.1*

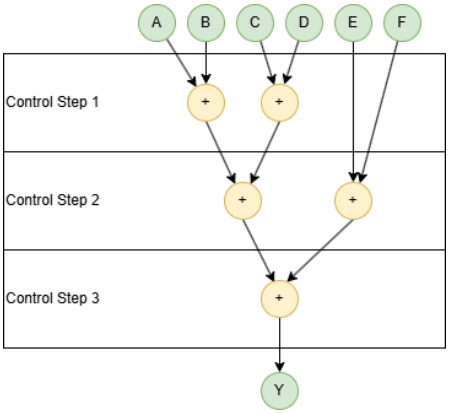


Figure 3.3: *ALAP schedule for 3.1*

tions to the UCS problem, they have limitations. They are insufficient for finding satisfactory solutions to the RCS and TCS problems, which require more advanced algorithms. [WC95]

3.2 Trace Sheduling

Joseph A. Fisher's 1981 paper, "Trace Scheduling: A Technique for Global Microcode Compaction" [Fis81], introduces an innovative compiler optimization technique to improve instruction-level parallelism, particularly for Very Long Instruction Word (VLIW) architectures. VLIW architectures are a type of computer processor design that seeks to exploit instruction-level parallelism by executing multiple operations simultaneously. Instead of having the processor figure out which instructions can run in parallel, a VLIW compiler does

this work ahead of time. In a VLIW system, the compiler bundles several independent instructions into one long word. Each part of this word corresponds to a different functional unit in the processor (like an adder, multiplier, etc.). When the VLIW processor reads this long word, it executes all the included operations simultaneously, assuming they don't depend on each other. The technique, known as trace scheduling, reorganizes instructions across multiple basic blocks, allowing compilers to optimize code beyond local (basic block-level) scheduling limitations. A basic block is a sequence of consecutive instructions in a computer program with a single entry point and a single exit point. This means that execution flows through the entire block without any possibility of branching (like conditional jumps) except at the end. The beginning of a basic block is marked by an instruction that is the target of a jump or branch, while the end is marked by a jump, branch, or a return statement. Basic blocks are important in compiler design because they allow for local optimizations and analyses, as the control flow is straightforward within each block.

The process begins with selecting a "trace", a linear sequence of basic blocks representing a frequently executed path through the program's control flow graph. This selection is typically informed by profiling or static analysis. Once a trace is identified, its instructions are rescheduled to exploit parallel execution opportunities while respecting hardware resource limits and data dependencies. To ensure that the optimized code remains correct even when execution diverges from the chosen trace, an additional "compensation code" is inserted at the entry and exit points of the trace.

However, the method is not free of limitations. The total code size might be increased by adding compensation code, and the performance advantages might differ depending on the real runtime behavior that does not closely match the predicted trace path. Despite these difficulties, trace scheduling remains a landmark work of compiler design and high-performance computing.

Applying this to the SCAD architecture allocation with the example of Figure 3.4, which depicts $y = x + x + x$, a problem arises. The figure represents a trace without a copy node and will be scheduled to one processing unit. For further consideration of this case, the following definitions will be used:

- $n_1 \rightarrow$ D will produce the values x_1 in PU.outL and x_2 in PU.outR while consuming x in PU.inR.
- $n_2 \rightarrow$ D will produce the values x_3 in PU.outL and x_4 in PU.inR while consuming x_1 in PU.inL.
- $n_3 \rightarrow$ AddN will produce the value x_5 in PU.outL while consuming x_3 in PU.inL x_4 in PU.inR.

Considering these definitions and a visualization of a Processing Unit Figure 3.6 and also defining mv as a move instruction that moves tokens from the output buffer to the corresponding input buffer, the following arises in the PU

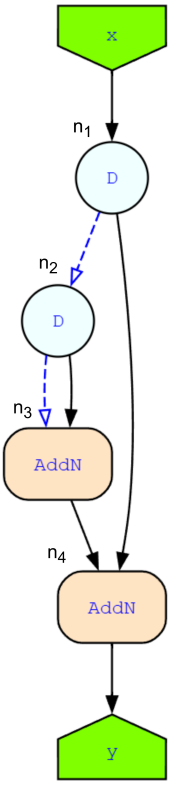


Figure 3.4: *Depiction of $y = x+x+x$ as a DPN with labeled nodes*

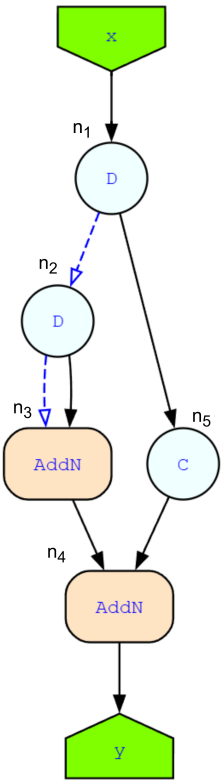


Figure 3.5: *Depiction of $y = x+x+x$ as a DPN with a copy node and labeled nodes*

Figure 3.7. Marked as red violates the strict FIFO behavior, representing a critical crossing; thus, this trace cannot be scheduled without adding a copy node. On the other hand, by adding a copy node as depicted in Figure 3.5 and defining the node as follows:

- $n_5 \rightarrow$ C will produce the value x_6 in PU.outL while consuming x_2 in PU.inR.

The visualization of this DPN executed on one Processing Unit with the addition of a copy node, as depicted in Figure 3.8, shows no violation of the strict FIFO behavior. Thus, this trace can be scheduled on one Processing Unit but only with the addition of another node, C, which will increase the execution time.

3.3 Vertex-Disjoint Paths

In scheduling, vertex-disjoint paths are a powerful abstraction for defining independent sequences of tasks that can be performed in parallel without conflicts. In a directed acyclic graph (DAG) $\mathcal{G}=(\mathcal{V},\mathcal{E})$, which is commonly used to model task dependencies in scheduling theory, each vertex $v \in \mathcal{V}$ denotes

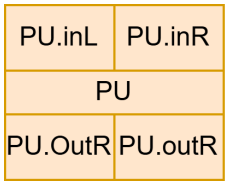


Figure 3.6: *Exemplary visualization of a Processing Unit*

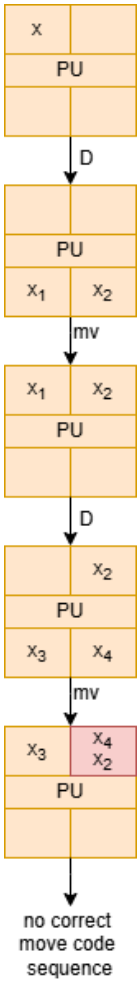


Figure 3.7: *Exemplary execution of the DPN 3.4 on one Processing Unit, showing a wrong buffer order, marked as red*

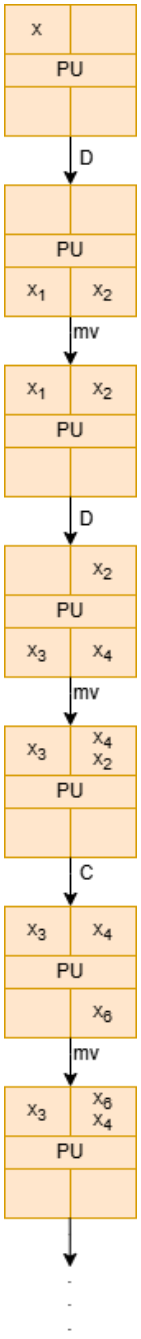


Figure 3.8: *Exemplary execution of the DPN 3.5 on one Processing Unit, with a correct buffer order, due to the added copy node*

a computational task, and each directed edge $(u,v) \in \mathcal{E}$ signifies a precedence constraint, meaning that task u must complete before task v begins. A set of vertex-disjoint paths in such a graph consists of paths which have no intermediate vertices; only the beginning and ending vertices may overlap if allowed. This guarantees that every task appears in the same path and can thus be mapped to a specific processing resource or execution slot.

In static scheduling of parallel programs, particularly on architectures with exposed datapaths or distributed execution units, vertex-disjoint paths enables the compiler or scheduler to assign entire chains of dependent tasks to dedicated processing elements. This method minimizes contention, avoids synchronization delays, and improves instruction-level parallelism (ILP). This approach is especially important for architectures such as SCAD (Synchronous Control Asynchronous Dataflow) and BED (Buffered Exposed Datapath), where static path allocation corresponds to queue-based scheduling. In [25b] Klaus Schneider point out that such path-based scheduling enables mapping DAG nodes to processing units with guaranteed ordering and no interference, thereby increasing throughput and hardware utilization.

The theoretical basis for using disjoint paths in graph decomposition comes from Menger's Theorem [Dit17]: the number of vertex-disjoint paths between nodes is less than the minimum vertex cut separating them. Although Menger's theorem offers a connectivity view, it is computationally exploited in scheduling, where the task graph is broken down into maximal sets of vertex-disjoint paths. In Leung's Handbook of scheduling [Leu04], he describes how path partitioning strategies make Scheduling easier and efficient mappings can be derived from task graphs to processor architectures.

Vertex-disjoint paths for scheduling have practical advantages: it enables local path selection, simplifies queue layout in FIFO-based dataflow machines, and enables scalable compilation. These aspects are especially critical in exposed datapath architectures where the communication and execution schedule is generated statically by the compiler, instead of dynamically at runtime.

3.4 Efficency of a Sheduling algorithm

A scheduling algorithm is usually evaluated for efficiency by analyzing how well it optimizes key performance metrics in a given computational environment. A metric presented in "Metrics and benchmarking for parallel job scheduling" [FR98] is the distribution of parallelism and runtime. Additionally, resource utilization measures the temporary allocation of the system's computational resources. Another key parameter should be the scheduling time, which represents the time and computational effort required by the algorithm to make scheduling decisions. However, this is not so important in statically scheduled processors since the schedule is calculated ahead of time, unlike in dynamically scheduled processors.

By analyzing how well the algorithm balances these factors, a conclusion can be made about the efficiency of a scheduling algorithm. Through examination of these aspects via simulation or empirical testing, assessments can be made

of how effectively a scheduling algorithm performs in practice.

3.5 Best possible Schedule for a DPN on a SCAD processor

The following section will explore developing the optimal schedule for a SCAD processor based on a DPN. This process will involve a systematic approach where we progressively impose restrictions on the available resources. By carefully analyzing these limitations, we will assess how they impact the efficiency of the generated schedule. Employing previously established metrics to evaluate the schedule's performance, ensuring that our assessment is thorough and encompasses the proposed planning framework's effectiveness and feasibility. Through this detailed examination, we aim to identify the best scheduling options and understand the trade-offs that come with resource constraints.

3.5.1 Ignoring limited recources and efficency

When ignoring limited resources and efficiency, finding a schedule for a given DPN architecture ($\mathcal{U}$, $\mathcal{C}$) comprising nodes $\mathcal{U}$ = {CU, IC, PU[0], ... , PU[n - 1]} and edges $\mathcal{C}$ and a Dataflow Graph ($\mathcal{P}$, $\mathcal{B}$), with nodes $\mathcal{P}$ interconnected by edges $\mathcal{B}$ is rather easy. Assuming the number of PUs, from now on referred to as #PU is greater or equal to the number of nodes in $\mathcal{P}$, then every node can be mapped to one PU. This results in an execution of the Dataflow Graph of the given DPN architecture without any possibility of critical crossings, since based on this PU allocation, the generated move instructions src $\rightarrow$ tgt either fall in the in chapter 2.5 established case 1 or 2.

This will result in PUs executing their node or instruction once the required data is transmitted over the IC. This approach is not feasible for real scheduling algorithms since the number of nodes scales with more complex computations. Even with the example of Figure 3.4, four PUs are used to calculate the result. However, with this approach, the best possible execution time of a program can be determined.

3.5.2 Considering restricting recources

When restricting resources, finding a schedule for a given DPN architecture ($\mathcal{U}$, $\mathcal{C}$) and a Dataflow Graph ($\mathcal{P}$, $\mathcal{B}$) is also moderately easy.

Assuming #PU less or equal to the number of nodes in $\mathcal{P}$, every trace can be mapped to one PU as long as no critical crossings are formed. This results in an execution of the dataflow graph of the given DPN architecture with, generally speaking, fewer Processing Units than in Section 3.5.1. The classification into traces is defined by each node $\mathcal{P}$ of the Dataflow Graph connected to two different nodes in $\mathcal{P}$, connected by the edges in $\mathcal{B}$, which are defined as two different traces, thus guaranteeing that no critical crossings are formed.

To illustrate the theoretical concepts outlined in the previous paragraph, let us consider a practical example demonstrating scheduling application within a DPN architecture. Specifically, we will examine a scenario involving a simple dataflow graph consisting of four independent traces as shown in Figure 3.9. This will result in PUs executing their traces sequentially while transmitting the resulting tokens over the IC to themselves.

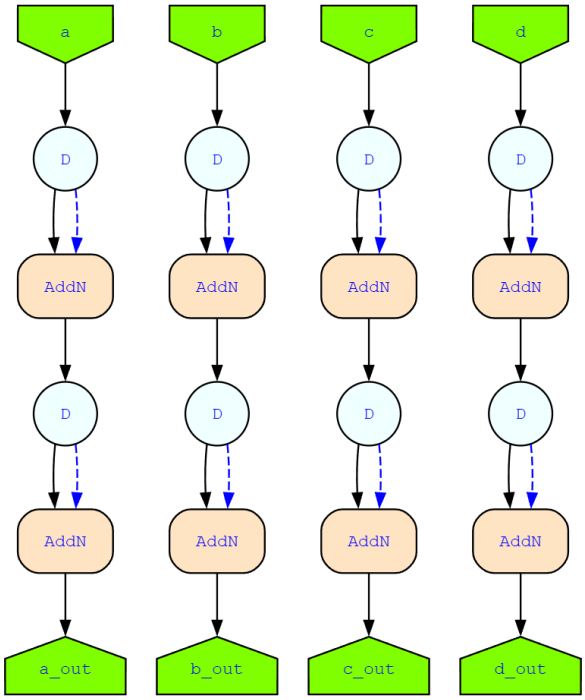


Figure 3.9: *A DPN consisting of four independent traces*

To illustrate another more realistic example, let us consider an intertwined DPN with Traces that are not independent. Specifically, we will examine a scenario involving a dataflow graph consisting of four traces, as shown in Figure 3.10. Here, trace 3 depends on traces 1 and 4, trace 1 depends on trace 3, trace 2 depends on trace 4, and trace 4 depends on trace 2. Trace 4 will execute when trace 2 produces the required token. Trace 3 will start executing when trace 1 produces the required token and hold until trace 4 produces the required token to keep executing the trace. Trace 1 will start executing directly and hold until trace 3 produces the required token to keep executing the trace. Trace 2 will start executing directly and hold until trace 4 produces the required token to keep executing the trace.

This approach is more feasible for real scheduling algorithms since the number of Processing Units scales with the number of traces instead of the number of nodes. Considering the example of Figure 3.4, the DPN depicted there can be divided into two traces as seen in Figure 3.11. Therefore, only 2 PUs will be used to calculate the result. However, with this approach, the best possible execution time of a program can still be determined, but the scheduling algo-

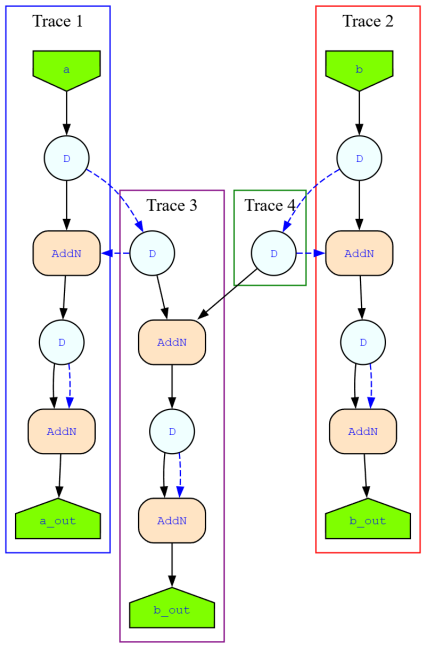


Figure 3.10: *A DPN consisting of 4 intertwined traces*

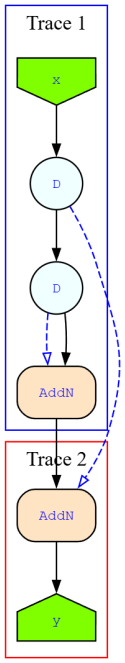


Figure 3.11: *Figure 3.4 divided into traces*

rithm needs to find all traces of the DPN, which makes the algorithm more complex.

3.5.3 Considering limited recources

As previously established in section 3.5.1 and section 3.5.3, the number of Processing Units needed for mapping a DPN to a DPN architecture was reduced from the number of nodes in a given DPN to the number of traces. However, now, considering mapping a DPN to a DPN architecture where #PU is less than the number of traces, which means that at least one processing unit needs to execute multiple traces. This is not always possible, as described in section 3.2 with the exemplary DPN shown in figure 3.4. The following paragraphs will ignore the impossible cases and only focus on cases that can be mapped to a DPN architecture where #PU is less than the number of taces. Nevertheless, while ignoring impossible cases, many possible cases, starting with the simplest and working up to the most difficult, still need to be addressed.

All traces are independent and of the same length

Since all traces are independent, they can be executed sequentially on any number of processing units, but since the best possible schedule for a DPN on a SCAD processor needs to be determined two simple cases arise that can be scheduled optimally as described in the following:

#PUs = 1 This is the easiest case; since only one processing unit is available, all traces need to be mapped onto this processing unit. The processing unit needs to execute the whole trace before executing another trace. The order of the traces does not matter. This will compromise an optimal schedule for mapping a DPN onto a SCAD processor, using only one processing unit.

#PUs mod #traces = 0 This is also relatively easy to schedule. Here, each processing unit is scheduled to execute the same number of traces; as explained before, all traces must be entirely executed before starting the execution of another trace. The order of the traces also does not matter. This will also compromise an optimal schedule since each processing unit bears the same load and can execute each trace in sequence.

All traces are independent but not of the same length

Since all traces are independent, they can be executed sequentially on any number of processing units. However, finding the best possible schedule for a DPN of this type on a SCAD processor is harder. Thus, only one straightforward case arises.

#PUs = 1 This case can be scheduled the same way as in the paragraph where traces have the same length. All traces need to be mapped to one processing unit; the execution of the whole trace needs to be ensured before starting the execution of another trace. This results in an optimal schedule for the DPN on one processing unit.

All other cases

Finding an optimal schedule for all the other cases is harder and cannot be described easily without establishing a new concept of scheduling these traces. The new concept that needs to be established will be called trace juggling and will be described in the following Section 3.5.4.

3.5.4 Trace Juggling

Before introducing the concept of Trace Juggling, let us consider a simple DPN consisting of four independent traces already beforehand, but now marked with labels $n_0 ... n_{15}$ for easier reference in Figure 3.12. This DPN must be scheduled onto an SCAD processor using three processing units. Applying the concepts of the previous section 3.5.3, we can assign each processing unit to execute one trace, which leaves one trace unassigned. If the unassigned trace is scheduled sequentially, two processing units will idle while one executes the whole trace. Splitting the nodes of the remaining trace to the different processing units will also not result in a faster execution since each node of the trace needs to be executed in sequence, meaning that in each step of the execution still, two processing units will idle.

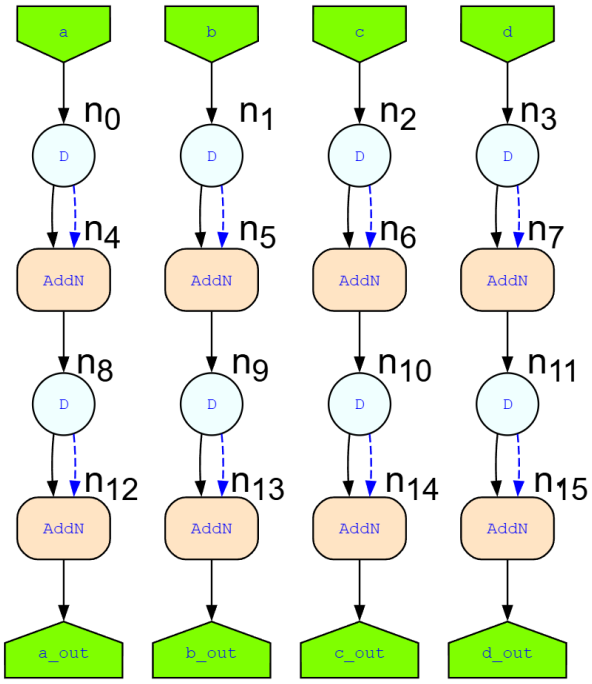


Figure 3.12: *Figure 3.9 with added labels for better referencing*

Now, consider the execution time of this schedule on a processing unit in clock cycles, which refers to a single tick of the processor's clock. For simplicity, every node will be executed in one clock cycle. The time it will take to transport the produced tokens to the subsequent buffers will be ignored. This will result in an execution time of 8 cycles since each trace takes 4 cycles, and one processing unit has to execute 2 traces in sequence.

However, what happens when not every trace is executed sequentially but is juggled between the processing units? Considering the following mapping of nodes onto the processing units and execution of the node based on the position in the following list:

- PU 0: $\{n_0, n_3, n_6, n_9, n_{12}, n_{15}\}$
- PU 1: $\{n_1, n_4, n_7, n_{10}, n_{13}\}$
- PU 2: $\{n_2, n_5, n_8, n_{11}, n_{14}\}$

Meanwhile, the nodes produce tokens in the form of nl_i and nr_i. The i will represent the number of the node producing the token, l will represent if the node produces a token in the left output buffer, and r will represent the right output buffer. Nodes n_{12}, n_{13}, n_{14}, and n_{15} will produce the corresponding outputs as depicted in Figure 3.12.

Considering the established mapping of nodes onto the processing units will result in the execution of nodes in the following cycles as visualized in 3.13:

- cycle 1: $\{n_0, n_1, n_2\}$

- cycle 2: $\{n_3, n_4, n_5\}$
- cycle 3: $\{n_6, n_7, n_8\}$
- cycle 4: $\{n_9, n_{10}, n_{11}\}$
- cycle 5: $\{n_{12}, n_{13}, n_{14}\}$
- cycle 6: $\{n_{15}\}$

Revisiting the previously established case of all traces being independent and of the same length and (PUs mod traces = 0), we can determine if trace juggling increases the execution time. As previously established, the optimal execution time can be achieved if each processing unit is scheduled to execute the same number of traces sequentially. Applying this to the DPN shown in Figure 3.12 with two processing units will result in an execution time of 8 cycles since each trace takes 4 cycles, and each processing unit will execute two traces.

However, when trace juggling, the mapping of nodes onto the processing units and execution of the node based on the position in the following list can be generated:

- PU 0: $\{n_0, n_2, n_4, n_6, n_8, n_{10}, n_{12}, n_{14}\}$
- PU 1: $\{n_1, n_3, n_5, n_7, n_9, n_{11}, n_{13}, n_{15}\}$

Considering the established mapping of nodes onto the processing units will result in the execution of nodes in the following cycles:

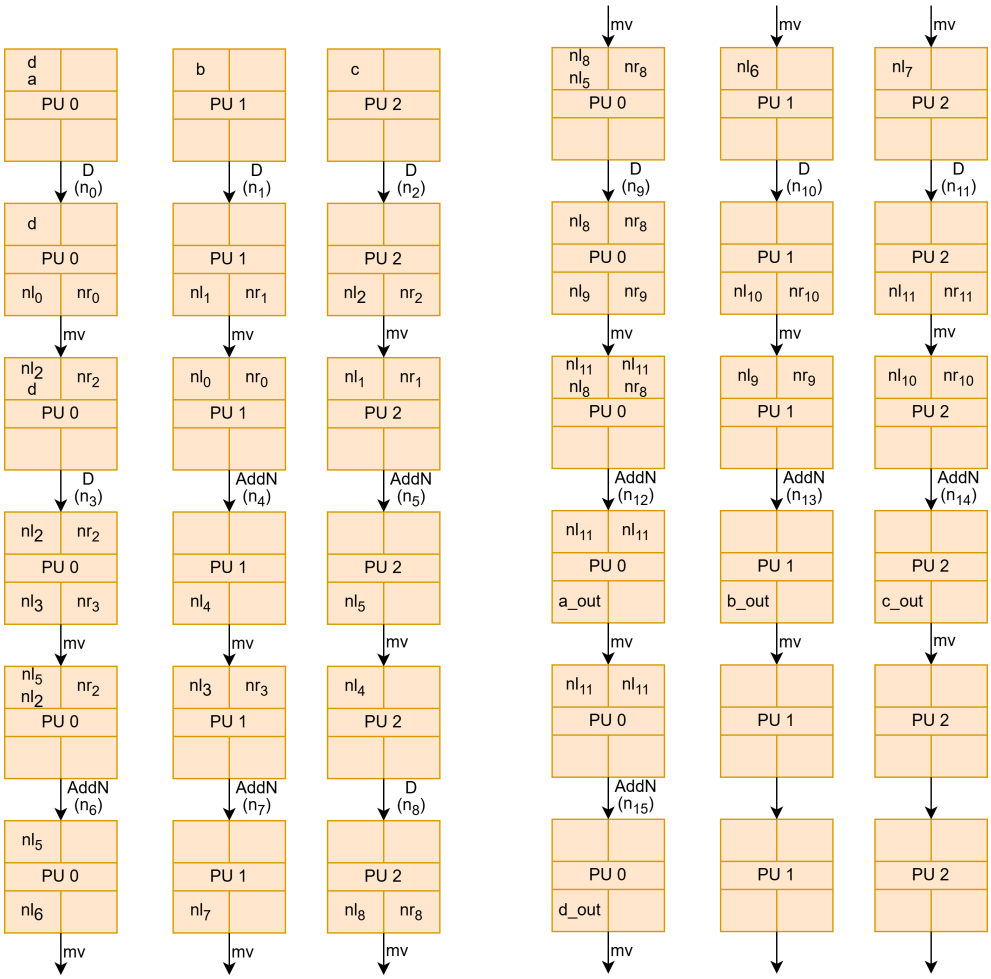
- cycle 1: $\{n_0, n_1\}$
- cycle 2: $\{n_2, n_3\}$
- cycle 3: $\{n_4, n_5\}$
- cycle 4: $\{n_6, n_7\}$
- cycle 5: $\{n_8, n_9\}$
- cycle 6: $\{n_{10}, n_{11}\}$
- cycle 7: $\{n_{12}, n_{13}\}$
- cycle 8: $\{n_{14}, n_{15}\}$

Using trace juggling will not create critical crossings for this case since each processing unit alternately executes nodes of two different traces while consuming the generated tokens and freeing up the buffer adhering to the strict FIFO structure. The execution of this schedule will take 8 cycles, thus not increasing the theoretical execution time.

For more complex DPN, including multiple intertwined traces, the evaluation is not easily shown and will be looked at in Section 5, while conducting experiments.

3.5.5 Summarizing Trace Juggling

To define Trace Juggling, a level must first be assigned to each node. This can be accomplished using the ALAP Scheduling algorithm. The contained nodes are assigned the level of the current control step number for each node. When assigning nodes to the Processing Units, the following must be considered. If another trace contains a node with a lower level that is not currently scheduled, and assigning it to this processing unit does not result in critical crossings, then it can be scheduled to this processing unit. In this case, the node with the smaller needs to be scheduled in the next step; otherwise, the next node in the trace can be assigned. A more detailed description of the application of this concept will be provided in Section 4.6.



.
.
.

Figure 3.13: *The visualization of the DPN depicted in figure 3.12, while using trace juggling*

4 Implementation

4.1 Averest

Averest offers a wide variety of tools intended for the specification, verification, and implementation of reactive systems [SS05] and is available as a NuGet package[1]. The implemented Scheduling algorithm will use the Averest Version 3.4.0. Within the Averest Namespace, a diverse range of compilers for synchronous languages, a simulator for those languages, formal verification support through temporal and other logics, and various transformations for synthesizing hardware and software for reactive embedded systems, which encompass both purely hardware circuits and purely software systems can be found [25a]. A couple of modules that will be used to implement a scheduling algorithm to extend allocations of processing units for Higher Instruction Level Parallelism can be found in the Averest.MiniC namespace in the CodeGenSCAD, MoveCode, IO and DataflowProcessNetworks modules[2]. These modules will be used in this chapter to translate MiniC into DPNs, generate a minimal allocation of processing units using SAT constraints, and finally translate the allocation into move code and simulate it to estimate a run time.

4.2 Establishing ground rules

Since the topic of this paper is "Extending Allocations of Processing Units for Higher Instruction Level Parallelism", a previously generated allocation of processing units for a given program will be considered. The generation of this allocation will be covered in Section 4.4. For extending these allocations, the following rules will be followed:

1. The previously generated allocation of processing units cannot be extended with other instructions, which means that the extension of the allocations will not map new nodes to the previously generated allocation.

2. The extended allocation is defined by the additional processing units assigned to the algorithm. If the number of additional processing units is zero, then the previously generated allocation will be returned. If the number of additional processing units is less than the number of processing units used by the previously generated allocation, the algorithm will hold and display an error.

[1] Averest NuGet package: https://www.nuget.org/packages/Averest
[2] Averest API Reference: http://www.averest.org/AverestLibDoc/reference/index.html

3. The extended allocations must be executable on a SCAD processor.
4. The extended allocation has to be executable faster, or as fast as the previously generated allocation.

After establishing these rules, an algorithm utilizing the previously defined concept in Chapter 3 can be applied to implement and evaluate an algorithm that extends the allocations of processing units for increased instruction level parallelism.

4.3 Establishing a folder structure

Before implementing functions to extend the allocation, a simple folder structure will be defined for more comfortable navigation. The DPN, located in the source directory, will contain all the necessary data required to generate an allocation, which is generated by the implemented functions. The DPN folder contains four subfolders containing the following:

- 1. Compile: contains two simple Python scripts used to generate Graph visualization
- ExampleMiniC: contains multiple subfolders with MiniC programs and the resulting data from the implemented functions. The MiniC programs here are the provided example functions on the averest website
- CustomMiniC: Also contains multiple subfolders with MiniC programs and the resulting data from the implemented functions. But here, all custom MiniC codes provided by the user of the scheduling algorithm will be stored
- MkRainbow: This contains multiple subfolders with DPN resulting from the MkRainbow function of Averest. The resulting data from the implemented functions will also be stored here.

4.4 Generating the underlying Allocations

As established in Section 4.2, the allocation of processing units needs to be generated before an algorithm can extend this allocation. To generate this allocation, multiple functions provided in Averest can be used in the generation:

ParseMiniCFromFile optOstr filename This function parses a program from a file, writing errors to the output stream 'optOstr' if it is not None and returning a MiniCProgram if no errors arise. The 'filename' describes the full path to the file.

MiniC2DPN syncCtrl mcp This translates a given MiniCProgram to an equivalent DPN, returning a DataflowProcessNetwork array. 'syncCtrl' can be ignored and will be set to false in the implementation; 'mcp' refers to the location of a MiniCProgram.

MkConstraintsSAT puNum optLevelInfo dpn This function generates constraints for code generation from dataflow graphs, returning a four-element sequence of values ('nodLevels', 'bufLevels', 'variables', 'constraints'). 'puNum' is the number of the PUs of the SCAD machine, 'optLevelInfo' should be a triple of the form (novel, bufLevel, maxNodLevel) option, but will be replaced in the implementation by None, and dpn is a DataflowProcessNetwork.

SolveConstraintsSAT puNum (nodLevels, bufLevels, variables, constraints) This function solves the constraints generated by MkConstraintsSAT, returning a four-element sequence of values ('nodOrder', 'bufOrder', 'nodAllocation', 'switchDupNodes') if the given constraints can be solved. 'bufOrder' and 'switchDupNodes' are igsnored; however, this was a fatal flaw and is examined more closely in Section 5.4. 'nodOrder' contains a sorted list of nodes per level, and 'nodAllocation' is a map that assigns each node of the DPN to one Processing Unit.

Using these four functions, the allocation of processing units can be generated. Furthermore, to generate an optimal starting point for the extension, a minimal allocation should be generated, which means that an allocation using a minimal number of processing units should generated. This can be achieved by sequentially increasing the number of processing units until the function SolveConstraintsSAT returns a four-element sequence of values. This is implemented in the new let MakeSAT function in the Logic module.

MakeSAT(name, subfolder) This function determines the minimum number of Processing units and the allocation required to execute a MiniC program on a SCAD processor, returning a four-element sequence of values ('minimalPUs', 'nodeOrder', 'nodeAllocation', 'dpn'). The input variables 'name' and 'subfolder' represent the filename of the MiniC program and the location of this file adhering to the folder structure previously established. 'minimalPUs' refers to the number of processing units required. 'nodeOrder' and 'nodeAllocation' are results from SolveConstraintsSAT, and 'dpn' represents a DataflowProcessNetwork.

4.5 Preparing the DPN

For the extension of the allocation, some preparation on the DPN needs to be addressed to facilitate the scheduling process. Therefore, three functions are implemented in the Logic module, and the name and the subfolder correspond to the location of the MiniC file that generated the DPN:

ParseFromDPN (name, subfolder) The function uses the Averests function PrintDPN, which writes a DPN to an output stream and transforms this output into an array of four-element sequences of values for each node. The sequence of values consists of the node number, the operation type, and the required tokens for the node's execution. The tokens are represented in the form x : v,

where x depicts the originating node of the token and v is the label used in the DPN output stream of the node. If x is depicted as '-', then the token is either a constant value, an input or an output.

FindLevels(name, subfolder) FindLevels uses ParseFromDPN to find a level for each node in a DPN, outputting levelized and levels. levelized contains an array with a subarray for each level. The subarrays are filled with all nodes that are sorted in the corresponding level. The output levels is an array of tuples where each node in the DPN a level is assigned.

FindTransitiveDependencies (name, subfolder) This function uses ParseFromDPN to find all dependencies of a node and returns a four-element sequence ('predecessor', 'successor', 'pOut', 'sOut'). 'predecessor' and 'successor' are arrays that contain a node and all transitive nodes that must be executed before and after the node's execution. 'pOut' and 'sOut' are arrays that contain all direct predecessors and successors that must be executed directly before and after the node.

The function FindLevels uses an algorithm based on ALAP Scheduling, as explained in Section 3.1, to assign a level to each node. This is used to ensure that tokens are only generated when needed and do not fill up the buffer of processing units unnecessarily. Using an algorithm based on ALAP Scheduling helps to reduce the creation of critical crossings and ensures faster execution times.

4.6 The Sheduling algorithm

To use the scheduler, the following function needs to be executed.

simpleShedule ('levelData', 'dependencies', 'satResults', 'pus', 'name', 'subfolder') This function's input is 'levelData', which is the result of FindLevels, 'dependencies', the result of FindTransitiveDependencies, and 'satResults', the result of MakeSAT. Additionally, 'pus' refers to the total number of processing units that should be used for the schedule. The variables 'name' and 'subfolder' refer to the name of the MiniC program and its location within the folder structure.

Using the scheduler is relatively easy, as only one function needs to be executed; however, the underlying algorithm that the function uses to find a schedule is more complex. The previously established concepts that are depicted in the Sections 3.2 and 3.5.4 are used to find an optimal schedule while still adhering to the established rules in Section 4.2.

The primary role of this scheduling function is to produce an efficient execution plan by leveraging results from a previous SAT-based solution, particularly in scenarios where the available number of PUs exceeds the minimum required.

Internally, the scheduling process is guided by a combination of dependency analysis and level information. Dependencies between operations are extracted and quantified in terms of the number of predecessors and successors for each node. This information is sorted and prioritized to help with scheduling decisions, especially for central nodes in the dependency graph. Central nodes are those that have many successors and should be prioritized so that the scheduler can assign operations more flexibly. The level data originating from the FindLevels function is then transformed into a node-level format to support actual scheduling and split up into possible processing units. This structure enables the function to track which operations belong to which processing unit at each logical time step or level of the computation. With this structure in place, the function initializes other processing unit objects for handling the extra PUs that are necessary. These objects store internal buffers and states for flexible scheduling techniques such as juggling (assigning new operations as defined in Section 3.5.4) or tracing (continuing the execution of operations in the buffer). The underlying function is an iterative scheduling loop. With each iteration, the scheduler attempts to allocate new operations to idle processing units (PUs) or to continue executing already buffered operations. The scheduling algorithm guarantees that all dependencies are respected and that operations assigned to PUs result in a valid execution sequence. This loop continues until all operations are scheduled and all buffers are cleared. When scheduling is completed, the function returns two primary outputs: the order of node execution and the node-to-processing unit mapping for each node.

In general, SimpleSchedule is a post-SAT scheduling layer that utilizes additional hardware resources for performance and parallelism. It integrates the results from previously established concepts to generate an execution plan for Dataflow Graphs.

4.7 Visualizing the results

A further visualization technique was designed to make the program execution on a SCAD processor accessible and easier to understand. This new approach enables users to understand the program flow and logic while the program executes, providing information about the execution steps. By expressing the program activities in a more natural manner, programmers and analysts may quickly find bottlenecks, improve performance, and also troubleshoot issues. This visualization technique helps with comprehension but also provides a more interactive experience since the user can see the changes and results made by the scheduling algorithm.

GraphvisGenerate(filename, path, dependencies, nodOrder, nodAllocation, puNum) This function takes the 'filename' and 'path', representing the location where the results will be stored. The dependencies variable resembles the result of the ParseFromDPN function. 'nodOrder', 'nodAllocation', and 'puNum' correspond to the generated schedule.

Using the GraphvisGenerate function will generate a Graphviz file that can be compiled into a PNG using the two simple Python scripts contained in folder 1. Compile. The visualization will follow the structure presented in Figure 4.1

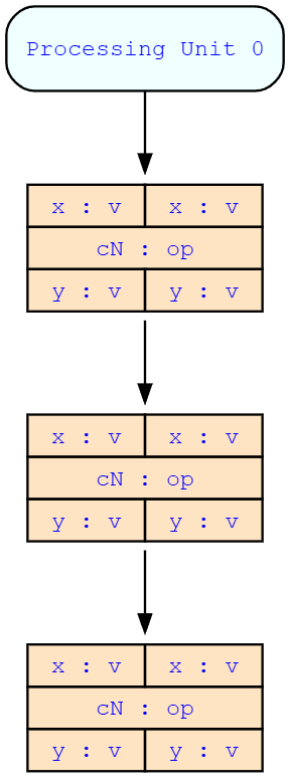


Figure 4.1: *The visualization of the resulting Graphviz file resulting from GraphvisGenerate*

The previously established Figure 3.6 inspired this representation in Figure 4.1. The variables x1 and x2 represent the nodes where the values vIn are generated. Furthermore, x1 and x2 represent the nodes that are defined as incoming edges of the DPN and need to be executed before the current node. The current node is represented by cN: op, where cN denotes the current node number that needs to be executed, and op denotes the operation that is executed. After executing the node cN, the values vOut are generated and sent to nodes y1 and y2, respectively, which are defined by the outgoing edges of the DPN. If a node produces only one token or only one token is used for the execution of the node, x2: vIn and y2: vIn will be empty. If a vIn is either a constant value or an input value, then x1 or x2 are represented as '-'. The same applies to the output values; thus, the corresponding y will be '-'.

4.8 Reducing Runtime

Generating an allocation using a minimal number of processing units is a time-intensive and lengthy process. Since the generation of the minimal number of processing units is provided by Averest functions, it cannot be optimized without implementing another function. To avoid long waiting times, especially when extending the PU allocation of one MiniC Program multiple times, the resulting generated allocation should be saved. The implemented function MakeSAT automatically saves the resulting allocation for a minimal number of processing units to a subfolder 'Minimal' in the folder of the MiniC program to speed up the process of extending the PU allocation. After the allocation of minimal processing units is generated the first time, the function reads the results saved in the text files ending with '.allocation' and '.minimal'. The file ending with '.minimal' contains the number of processing units that are at least required to generate an allocation, while '.allocation' contains the node Order and node Allocation.

4.9 Using the Schedular

Using the implemented scheduler is relatively easy. When starting the scheduler, a console application guides the user through the process of moving the MiniC program to the correct folder based on the established folder structure, selecting the correct MiniC program to be scheduled, and warning the user if an error occurs.

5 Experimental evaluation

In Section 5.1, a overview of the schedulers utilized for the evaluation is provided. This section outlines the criteria for their selection, detailing the unique features and capabilities that make them suitable for our research objectives. Additionally, we will elaborate on the experimental methodology employed, including the specific parameters and settings under which the evaluations were conducted, as well as the rationale behind our chosen approach. This thorough explanation aims to ensure clarity and reproducibility in our findings.

To make this comparison more tangible, we will anchor our discussion in a specific example that exemplifies these scheduling mechanisms. This approach will not only clarify the theoretical concepts but also demonstrate their practical relevance in real-world scenarios.

Furthermore, this Section 5.2 will delve into the intricate relationship between the allocation of processing units and the effectiveness of the extended schedule. Ultimately, our analysis will show that the choice of scheduler, in conjunction with the way processing units are allocated, plays a crucial role in determining the success and efficiency of the overall scheduling strategy.

Finally, we will examine additional examples to gain a deeper understanding of how the scheduling algorithm impacts the execution time. This will involve analyzing various scenarios in which the algorithm has been applied, allowing us to assess its efficiency, adaptability, and performance under different conditions. By doing so, we aim to identify best practices and potential improvements that can enhance its functionality.

5.1 Expirimental Setup

In embedded systems, scheduling algorithms must be optimized for performance and resource utilization. Three different scheduling techniques that are important for evaluating the implemented scheduler will be discussed: constructing a Schedule for a minimum number of processing units (described in Section 4.4), optimal scheduling (defined in Section 3.5.1), and vertex-disjoint paths (depicted in Section 3.3). Each of these methodologies provides different perspectives and methods for task management in embedded systems. By critically analyzing the execution time of these techniques, we highlight their efficiency and generalizability for different applications.

A SCAD simulator can be used as a key tool for testing the systems. The values of the inputs for each DPN are replaced with the constant value of 0. This isolates components and verifies functionality independent of variable inputs.

After the scheduling is complete and the resulting schedule is translated into

move code to translate the given allocation, the Averest function CodeGenScad can be used. Simulating the code will determine the number of cycles required to execute it.

Two methods are available to simulate the code. The first is to use the provided Averest function, which simplifies the simulation and provides an idea about the code's performance. The simulations can also run from the Teaching Tools website[1]. This platform has a basic user interface and visual representation of the execution cycle. Both methods are useful for analyzing the code's operational efficiency and execution time in cycles.

5.2 Looking deeper into one specific example

The following code was developed by the Embedded Systems Group at Rheinland-Pfalz Technical University (RPTU) in Kaiserslautern. It will be the main course of discussion for evaluating the implemented software in depth.

Listing 5.1: *MinTwoPusNeeded MiniC program*

```
// ------------------------------------------------------------------------
// This MiniC program generates a DPN that requires two PUs for scheduling
// even if we allow that the outputs of D-nodes are switched.
// ------------------------------------------------------------------------

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

    thread t105 {
        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;
    }
```

This MiniC program specifies a basic calculation to demonstrate a situation in which at least two Processing Units are required to schedule. On the upper level, the program specifies global variables x00 to x03 as input variables and x30 to x33 as the calculated outcomes. All intermediate computations are performed in thread t105 using local variables.

In the first computation layer within the thread, intermediate variables x10 through x13 are calculated based on the global inputs. These are independent and can be performed in parallel.

[1] Teaching Tools website for simulating move code: https://es.cs.rptu.de/tools/teaching/ScadSim.html

The second layer is based on the first, and variables x20 through x23 are calculated from the first layer. Some expressions use values from several first-layer variables; for example, both x20 and x23 require x10 and x13, so there is some data contention there. Other expressions, such as x21 = x11 + x11, repeat the use of the same variable, exhibiting self-dependency without introducing new external dependencies. Finally, the third layer of computation returns the values x30 through x33 from the second layer. These outputs have overlapping dependencies; for instance, x33 and x30 both need x23 and x20, while x31 and x32 require x22 and x21. This dependency overlap implies that some computations must wait till their inputs are obtainable, and some intermediate results have to be obtained more than once, further complicating the scheduling problem. This program presents a challenging program scheduling challenge due to its data reuse and dependency structure. Even if the output execution order is flexible, the overlapping use of intermediate values makes some computations wait for others to finish. If all operations are mapped to a single processing unit (PU), this leads to contention. The scheduler must distribute tasks among at least two processing units to keep tasks progressing and prevent execution stalls.

The previous program can be visualized as a DPN, resulting in Figure 5.1. Additionally, all nodes can be assigned levels based on their position in the DPN due to the dependencies established by the incoming and outgoing edges in the graph. Levels can be thought of as hierarchical layers that reflect the order of dependency among the nodes. Nodes with lower levels may serve as parents for nodes at higher levels because nodes of a higher level consume the tokens of lower levels and can, therefore, only be executed when the nodes of the lower levels are executed beforehand. With this in mind, the DPN can be divided into the following levels:

- level 1 {12, 13, 22, 23}
- level 2 {0, 7, 3, 6}
- level 3 {14, 15, 16, 17}
- level 4 {1, 2, 4, 5}
- level 5 {18, 19, 20, 21}
- level 6 {8, 9, 10, 11}

As previously established, the minimum number of processes required is already defined as two. Using the SatConstraints to find a schedule results in the following node order and Pu allocation

- PU 0 {1, 3, 5, 6, 7, 9, 15, 16, 17, 19, 20, 21}
- PU 1 {0, 2, 4, 6, 8, 10, 11, 12, 13, 14, 18, 22, 23}

The following node order:
{12, 13, 22, 23, 7, 6, 16, 17, 2, 20, 4, 21, 10, 3, 9, 0, 15, 14, 1, 18, 5, 19, 8, 11}

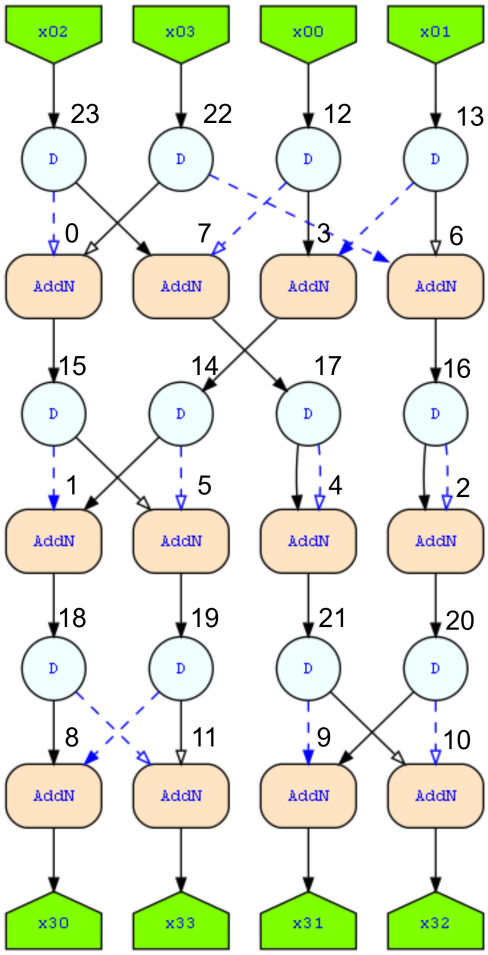


Figure 5.1: *The visualization of DPN of the Program MinTwoPusNeeded, labeled with the number of each node according to the DPN*

Transforming this schedule into move code and replacing the input variables x00, x01, x02, and x03 with constant values to factor out the time required for memory access, the execution of this schedule will take 18 cycles. This is far from optimal, as, for example, Processing Unit 1 handles all the token duplications in level 1; thus, processing unit 0 stalls until cycle 3 to execute instruction 7, which is the first instruction scheduled to be executed on pu 0.

Using the implemented scheduling algorithm to extend the allocations to use other processing units results in the following schedule:

- PU 0 { 3, 5, 7, 9, 16, 17, 19}
- PU 1 {4, 6, 8, 10, 11, 14, 18, 23}
- PU 2 {0, 1, 2, 12, 13, 15, 20, 21, 22}

The following node order:
{23, 12, 13, 22, 7, 6, 0, 3, 17, 14, 15, 16, 5, 4, 2, 1, 19, 18, 20, 21, 9, 10, 8, 11}

Sheduling Algorithm	PUs used	Execution time in cycles.
optimal Schedule	24	6
minimal PUs	2	18
Vertex-Disjoint Paths	4	6
Extended Schedule 3	3	12
Extended Schedule 4	4	9
Extended Schedule 5	5	6

Table 5.1: *Execution time for MinTwoPusNeeded with different scheduling approaches*

Transforming this schedule into move code and also replacing the input variables x00, x01, x02, and x03 with constant values to factor out the time required for memory access, the execution of this schedule will take 12 cycles. This schedule reduces the execution time from 18 to 12 cycles by adding only one processing unit. This is still not close to an optimal schedule. As mentioned before, processing unit 0 does not handle any nodes level 1. This is due to the established rules in Section 4.2 and cannot be changed.

Continuing to expand this allocation results in a further decrease in the number of cycles needed for the execution of this DPN on more processing units. Furthermore, as shown in Table 5.1, the minimal number of cycles needed for the execution of this DPN is reached with five processing units. This allocation reaches the minimum number of cycles for execution, but the algorithm Vertex-Disjoint Paths uses only four processing units to achieve the same result.

5.3 Expanding the examples for further evaluation

Looking deeper into other DPNs to further evaluate the efficiency of the developed scheduling algorithm. The special DPN, called the Rainbow DPN, is a DPN that arises from the calculation y = (x + 1) * ... * (x + n). Using the implemented Scheduler to evaluate potential efficiency yields the following results, as depicted in Tables 5.2 and 5.3. The implemented Scheduler reaches an optimal execution time by using fewer processing units than the algorithm using Vertex-Disjoint Paths. The Rainbow DPN with n = 3 utilizes one fewer processing unit than the Vertex-Disjoint Paths yet achieves an optimal execution time of 13 cycles. For n = 4, the Scheduler can utilize two fewer processing units, thereby increasing the execution cycles by 1. By using one fewer processing unit, the algorithm achieves an optimal execution time of 19 cycles.

Rainbow 3		
Sheduling Algorithm	PUs used	Execution time in cycles.
optimal Shedule	10	13
Vertex-Disjoint Paths	3	13
Extended Shedule 3	2	13

Table 5.2: *Execution time for make Rainbow 3 with different scheduling approaches*

Rainbow 4		
Scheduling Algorithm	PUs used	Execution time in cycles.
optimal Schedule	14	19
Vertex-Disjoint Paths	5	19
Extended Schedule 3	3	20
Extended Schedule 4	4	19

Table 5.3: *Execution time for make Rainbow 4 with different scheduling approaches*

5.4 The failure of the algorithm

While executing, a critical oversight arose: the previous allocation for a minimal number of processing units considered switching the results of duplication nodes. As stated in the implementation, switching the results of duplication nodes could be ignored; however, this resulted in an error while translating the allocation to move code. Since the underlying allocation of minimal processing units utilizes switched outputs for generating the allocation for minimal processing units, the dependencies vary drastically from the original DPN. Ignoring the switched Duplication nodes results in the schedule not being executable anymore, thus giving the algorithm a wrong starting point and, therefore, dooming the implemented scheduler to failure.

6 Conclusion

This thesis conducted a systematic study of scheduling algorithms for dataflow process networks (DPNs) with a special emphasis on execution time optimization via efficient processing unit (PU) allocation. In a series of carefully constructed experiments, we demonstrated how scheduling strategies—ranging from minimal PU allocation to extended schedules and vertex-disjoint path methods—impact execution performance in various scenarios.

The MinTwoPusNeeded example demonstrated that even relatively simple programs can reveal very complex scheduling dependencies that require parallel execution. Our evaluation revealed that exceeding the minimum required number of processing units can significantly reduce execution time, highlighting the tradeoff between resource utilization and performance.

Further testing with Rainbow DPNs confirmed these results, showing that the implemented scheduler can, in some cases, outperform more established methods by achieving optimal or near-optimal cycle counts with reduced resources. These results validate the general principles of the implemented algorithm and its generalization to different dependency structures in DPNs.

This study also revealed important limitations. Most importantly, improper handling of switched outputs in duplication nodes caused misalignment of dependency graphs and eventually invalidated some generated schedules. This highlights the need for consistency between theoretical models and practice, especially when transformations or optimizations related to node behavior or connectivity are applied.

Although the proposed scheduling algorithm reveals promising performance characteristics and scalability, it is necessary to refine it to handle edge cases and ensure that the implemented scheduler functions with any DPN.

7 Future Work

Although the proposed scheduler developed in this thesis produces encouraging results, several areas of work remain to be investigated, which may substantially enhance its performance, generality, and robustness. The immediate priority is the handling of duplication node semantics in cases of switched outputs. The current implementation does not account for these differences fully and can result in discrepancies between the theoretical Dataflow Process Network (DPN) and the generated move code.

Performance-wise, future improvements are possible. As shown in Section 5.2, the underlying allocation is not optimal. It should be regenerated using another algorithm to accommodate the number of processing units considered directly, thereby generating a more optimal schedule from the outset.

List of Figures

Bibliography

[25a] *Averest, Model-based Design of Reactive Embedded Systems*. Accessed: 01-June-2025. 2025. URL: http://www.averest.org/.

[25b] *Exposed Datapath Architectures*. Accessed: 01-June-2025. 2025. URL: https://es.cs.rptu.de/teaching/procarch/slides/PA-10-ExposedDatapath-1.pdf.

[ABS18] Markus Anders, Anoop Bhagyanath, and Klaus Schneider. "On memory optimal code generation for exposed datapath architectures with buffered processing units". In: *2018 18th International Conference on Application of Concurrency to System Design (ACSD)*. IEEE. 2018, pp. 115–124.

[AC97] Marnix Arnold and Henk Corporaal. "Data transport reduction in move processors". In: *Third Annual Conference of the Advance School for Computing and Imaging*. Citeseer. 1997, pp. 68–75.

[Aga99] Anant Agarwal. "Raw Computation". In: *Scientific American* 281.2 (1999), pp. 60–63. ISSN: 00368733, 19467087. URL: http://www.jstor.org/stable/26058367 (visited on 06/01/2025).

[Bha21] Anoop Bhagyanath. "Code generation for synchronous control asynchronous dataflow architectures". PhD thesis. Dissertation, Kaiserslautern, Technische Universität Kaiserslautern, 2020, 2021.

[BS16] Anoop Bhagyanath and Klaus Schneider. "Optimal compilation for exposed datapath architectures with buffered processing units by SAT solvers". In: *2016 ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. IEEE. 2016, pp. 143–152.

[BS17] Anoop Bhagyanath and Klaus Schneider. "Exploring the Potential of Instruction-Level Parallelism of Exposed Datapath Architectures with Buffered Processing Units". In: *2017 17th International Conference on Application of Concurrency to System Design (ACSD)*. 2017, pp. 106–115. DOI: 10.1109/ACSD.2017.20.

[BSS14] Nikita Bhardwaj, Maximilian Senftleben, and Klaus Schneider. "Abacus: A Processor Family for Education". In: *Proceedings of the WESE'14: Workshop on Embedded and Cyber-Physical Systems Education*. WESE'14. New Delhi, India: Association for Computing Machinery, 2014. ISBN: 9781450330909. DOI: 10.1145/2829957.2829959. URL: https://doi.org/10.1145/2829957.2829959.

[Bur+04] Doug Burger, Stephen W Keckler, Kathryn S McKinley, Mike Dahlin, Lizy K John, Calvin Lin, Charles R Moore, James Burrill, Robert G McDonald, and William Yoder. "Scaling to the End of Silicon with EDGE Architectures". In: *Computer* 37.7 (2004), pp. 44–55.

[CJA00] Henk Corporaal, Johan Janssen, and Marnix Arnold. "Computation in the context of transport triggered architectures". In: *International Journal of Parallel Programming* 28 (2000), pp. 401–427.

[CL95] Henk Corporaal and Reinoud Lamberts. "TTA processor synthesis". In: *First Annual Conf. of ASCI*. Citeseer. 1995, pp. 18–27.

[Cor94] Henk Corporaal. "Design of transport triggered architectures". In: *Proceedings of 4th Great Lakes Symposium on VLSI*. IEEE. 1994, pp. 130–135.

[Cor99] Henk Corporaal. "TTAs: Missing the ILP complexity wall". In: *Journal of Systems Architecture* 45.12-13 (1999), pp. 949–973.

[Dit17] Christoph Dittmann. "Menger's Theorem". In: *Archive of Formal Proofs* (2017).

[Fis81] Fisher. "Trace Scheduling: A Technique for Global Microcode Compaction". In: *IEEE Transactions on Computers* C-30.7 (1981), pp. 478–490. DOI: 10.1109/TC.1981.1675827.

[FR98] Dror G. Feitelson and Larry Rudolph. "Metrics and benchmarking for parallel job scheduling". In: *Job Scheduling Strategies for Parallel Processing*. Ed. by Dror G. Feitelson and Larry Rudolph. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 1–24. ISBN: 978-3-540-68536-4.

[GH05] Stephan Gatzka and Christian Hochberger. "The AMIDAR class of reconfigurable processors". In: *the Journal of Supercomputing* 32 (2005), pp. 163–181.

[Gov+12] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. "Dyser: Unifying functionality and parallelism specialization for energy-efficient computing". In: *IEEE Micro* 32.5 (2012), pp. 38–51.

[HC92] Jan Hoogerbrugge and Henk Corporaal. "Comparing software pipelining for an operation-triggered and a transport-triggered architecture". In: *Compiler Construction: 4th International Conference, CC'92 Paderborn, FRG, October 5–7, 1992 Proceedings 4*. Springer. 1992, pp. 219–228.

[HC94a] Jan Hoogerbrugge and Henk Corporaal. "Register file port requirements of transport triggered architectures". In: *Proceedings of the 27th annual international symposium on Microarchitecture*. 1994, pp. 191–195.

[HC94b] Jan Hoogerbrugge and Henk Corporaal. "Transport-triggering vs. operation-triggering". In: *International Conference on Compiler Construction*. Springer. 1994, pp. 435–449.

[Hoo97] Jan Hoogerbrugge. "Code generation for transport-triggered architectures." In: (1997).

[Jää+18] Pekka Jääskeläinen, Aleksi Tervo, Guillermo Payá Vayá, Timo Viitanen, Nicolai Behmann, Jarmo Takala, and Holger Blume. "Transport-triggered soft cores". In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2018, pp. 83–90.

[KC97] Ireneusz Karkowski and Henk Corporaal. "Design of heterogenous multi-processor embedded systems: applying functional pipelining". In: *Proceedings 1997 International Conference on Parallel Architectures and Compilation Techniques*. IEEE. 1997, pp. 156–165.

[Ker23] Nadine Kercher. "Code Generation for Buffered Exposed Datapath Architectures". In: (2023).

[Lei+20] Charles E. Leiserson, Neil C. Thompson, Joel S. Emer, Bradley C. Kuszmaul, Butler W. Lampson, Daniel Sanchez, and Tao B. Schardl. "There's plenty of room at the Top: What will drive computer performance after Moore's law?" In: *Science* 368.6495 (2020), eaam9744. DOI: 10.1126/science.aam9744. eprint: https://www.science.org/doi/pdf/10.1126/science.aam9744. URL: https://www.science.org/doi/abs/10.1126/science.aam9744.

[Leu04] Joseph YT Leung. *Handbook of scheduling: algorithms, models, and performance analysis*. Chapman and Hall/CRC, 2004.

[Moo65] Gordon Moore. "Moore's law". In: *Electronics Magazine* 38.8 (1965), p. 114.

[San+03] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W Keckler, and Charles R Moore. "Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture". In: *Proceedings of the 30th annual international symposium on Computer architecture*. 2003, pp. 422–433.

[San+04] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Nitya Ranganathan, Doug Burger, Stephen W Keckler, Robert G McDonald, and Charles R Moore. "Trips: A polymorphous architecture for exploiting ilp, tlp, and dlp". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 1.1 (2004), pp. 62–93.

[SB23] Klaus Schneider and Anoop Bhagyanath. "Consistency Constraints for Mapping Dataflow Graphs to Hybrid Dataflow/von Neumann Architectures". In: *ACM Transactions on Embedded Computing Systems* 22 (July 2023). DOI: 10.1145/3607869.

[SBR22a] Klaus Schneider, Anoop Bhagyanath, and Julius Roob. "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 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 133–145. ISBN: 9781450392662. DOI: 10.1145/3519941.3535076. URL: https://doi.org/10.1145/3519941.3535076.

[SBR22b] Klaus Schneider, Anoop Bhagyanath, and Julius Roob. "Virtual Buffers for Exposed Datapath Architectures". In: *MBMV 2022; 25th Workshop*. 2022, pp. 1–11.

[Sch21] Klaus Schneider. "Translating structured sequential programs to dataflow graphs". In: *Proceedings of the 19th ACM-IEEE International Conference on Formal Methods and Models for System Design*. MEMOCODE '21. Virtual Event, China: Association for Computing Machinery, 2021, pp. 66–77. ISBN: 9781450391276. DOI: 10.1145/3487212.3487343. URL: https://doi.org/10.1145/3487212.3487343.

[SS05] Klaus Schneider and Tobias Schuele. "Averest: Specification, verification, and implementation of reactive systems". In: *Conference on Application of Concurrency to System Design (ACSD)*. Citeseer. 2005.

[Tay+02] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, et al. "The raw microprocessor: A computational fabric for software circuits and general-purpose programs". In: *IEEE micro* 22.2 (2002), pp. 25–35.

[Tay99] Michael Bedford Taylor. "Design decision in the implementation of a raw architecture workstation". PhD thesis. Massachusetts Institute of Technology, 1999.

[Wai+97] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, et al. "Baring it all to software: Raw machines". In: *Computer* 30.9 (1997), pp. 86–93.

[Wai04] E Waingold. "Baring it all to software: Raw machines". In: *IEEE Trans. on Comput.* 53.11 (2004), pp. 1436–1448.

[WC95] R.A. Walker and S. Chaudhuri. "Introduction to the scheduling problem". In: *IEEE Design Test of Computers* 12.2 (1995), pp. 60–69. DOI: 10.1109/54.386007.