Imperial College London
Department of Electrical and Electronic Engineering

Combining Dynamic and Static Scheduling in High-Level Synthesis

Jianyi Cheng

June 9, 2023

Supervised by
Prof. George A. Constantinides
Dr John Wickerson

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Electrical and Electronic Engineering of Imperial College London
and the Diploma of Imperial College London

© The copyright of this thesis rests with the author. Unless otherwise indicated, its contents are licensed under a Creative Commons Attribution-Non Commercial 4.0 International Licence (CC BY-NC).

Under this licence, you may copy and redistribute the material in any medium or format. You may also create and distribute modified versions of the work. This is on the condition that: you credit the author and do not use it, or any derivative works, for a commercial purpose.

When reusing or sharing this work, ensure you make the licence terms clear to others by naming the licence and linking to the licence text. Where a work has been adapted, you should indicate that the work has been changed and describe those changes.

Please seek permission from the copyright holder for uses of this work that are not included in this licence or permitted under UK Copyright Law.

Abstract

Field Programmable Gate Arrays (FPGAs) are starting to become mainstream devices for custom computing, particularly deployed in data centres. However, using these FPGA devices requires familiarity with digital design at a low abstraction level. In order to enable software engineers without a hardware background to design custom hardware, high-level synthesis (HLS) tools automatically transform a high-level program, for example in C/C++, into a low-level hardware description.

A central task in HLS is scheduling: the allocation of operations to clock cycles. The classic approach to scheduling is static, in which each operation is mapped to a clock cycle at compile-time, but recent years have seen the emergence of dynamic scheduling, in which an operation's clock cycle is only determined at run-time. Both approaches have their merits: static scheduling can lead to simpler circuitry and more resource sharing, while dynamic scheduling can lead to faster hardware when the computation has non-trivial control flow.

This thesis proposes a scheduling approach that combines the best of both worlds. My idea is to use existing program analysis techniques in software designs, such as probabilistic analysis and formal verification, to optimize the HLS hardware. First, this thesis proposes a tool named DASS that uses a heuristic-based approach to identify the code regions in the input program that are amenable for static scheduling and synthesises them into statically scheduled components, also known as static islands, leaving the top-level hardware dynamically scheduled. Second, this thesis addresses a problem of this approach: that the analysis of static islands and their dynamically scheduled surroundings are separate, where one treats the other as black boxes. We apply static analysis including dependence analysis between static islands and their dynamically scheduled surroundings to optimize the offsets of static islands for high performance. We also apply probabilistic analysis to estimate the performance of the dynamically scheduled part and use this information to optimize the static islands for high area efficiency. Finally, this thesis addresses the problem of conservatism in using sequential control flow designs which can limit the throughput of the hardware. We show this challenge can be solved by formally proving that certain control flows can be safely parallelised for high performance. This thesis demonstrates how to use automated formal verification to find out-of-order loop pipelining solutions and multi-threading solutions from a sequential program.

Acknowledgements

I would like to express my deepest appreciation to my supervisors Prof. George A. Constantinides and Dr John Wickerson. George has been guiding me since my master course with his deep insights, especially in the world of high-level synthesis, which resulted in this thesis. He also taught me how to generalise a practical problem and formalise it using mathematical models. John has been supporting my research all along with his wide-ranging knowledge. He opened the door for me to the programming language research community and significantly improved my skills on expressing research ideas, both written and presentation. They have both been amazing supervisors and provided me with a wonderful PhD experience. It has been a privilege working with them.

I also would like to thank my PhD examiners, Prof. Paul Kelly and Prof. Zhiru Zhang for the insightful discussions we had during my PhD viva and their helpful feedback on improving the quality of this thesis.

I would like to extend my sincere thanks to my external collaborators, especially Prof. Paolo Ienne at EPFL and Dr Lana Josipović at EPFL (now at ETH). They introduced me to the world of dynamic scheduled high-level synthesis, leading to my first PhD publication. Lana has also been continuously working with me and provided me with helpful insights.

I also had the great pleasure of working with other collaborators, especially Dr Shane Fleming (now at AMD Xilinx), Dr Ruizhe Zhao (now at DeepMind) and Prof. Wayne Luk at Imperial College London, Dr Stephen Neuendorffer at AMD Xilinx, and Prof. Jason Anderson at the University of Toronto. My success would not have been possible without their help.

My PhD experience would not be complete without the support of my friends. I would like to sincerely thank all my PhD colleagues, who have accompanied me on this journey - two Bens, Sina, Erwei, Yann, Dan, Alex, Zhewen, Adi, Deetz, Sam and the rest of the research group.

Finally, I am grateful for the endless support given by my girlfriend Hao and my parents. Their sacrifices allow me to strive for excellence and achieve my dream.

Dedication

This work is dedicated to my dearest parents as well as my beloved girlfriend, Hao, without whose continuous support and encouragement this thesis would not be possible.

"People who are really serious about software should make their own hardware."

Alan Kay

Contents

Copyright Statement **i**

Abstract **iii**

Acknowledgements **v**

1 Introduction **1**

1.1 Research Contributions 4

1.2 Thesis Outline 5

1.3 Statement of Originality 7

2 Background **9**

2.1 Field Programmable Gate Array 9

2.2 High-Level Synthesis 11

2.2.1 HLS Tool Framework 12

2.2.2 HLS Tool Flow 16

2.2.3 HLS Passes for Hardware Optimisation 17

2.3 Scheduling 19

2.3.1 Static Scheduling 20

2.3.2 Dynamic Scheduling 21
2.3.3 Static and Dynamic Scheduling 24
2.4 Static Analysis for HLS 24
2.4.1 Polyhedral Techniques 25
2.4.2 Formal Verification 25
2.4.3 Probabilistic Analysis 27
2.4.4 Comparison 27
2.5 Benchmarks 28
2.6 Summary 32

3 Scheduling Overview and DASS Tool 33

3.1 Scheduling Formalisation 33
3.1.1 Scheduling Specifications 33
3.1.2 Control Flow Graph 34
3.1.3 Static Scheduling Implementation 36
3.1.4 Dynamic Scheduling Implementation 38
3.1.5 Optimisations for Dynamic & Static Scheduling 41
3.2 DASS HLS Tool 44
3.2.1 Introduction 45
3.2.2 Motivating Example 46
3.2.3 Methodology 50
3.2.4 Experiments 62
3.2.5 Case Study 1: II Exploration 67

3.2.6 Case Study 2: BubbleSort . . . 72

3.3 Summary . . . 74

4 Finding and Balancing Static Islands 75

4.1 Finding Static Islands . . . 75

4.1.1 Introduction . . . 76

4.1.2 Motivating Example . . . 78

4.1.3 Background . . . 82

4.1.4 Methodology . . . 82

4.1.5 Experiments . . . 98

4.1.6 Summary . . . 102

4.2 Balancing Static Islands . . . 103

4.2.1 Introduction . . . 103

4.2.2 Motivating Example . . . 105

4.2.3 Background . . . 108

4.2.4 Methodology . . . 109

4.2.5 Tool Flow . . . 123

4.2.6 Experiments . . . 125

4.2.7 Summary . . . 128

5 Parallelising Control Flow 129

5.1 Dynamic C-slow Pipelining . . . 130

5.1.1 Introduction . . . 130

5.1.2 Motivating Example . . . 132

5.1.3 Background . . . 134
5.1.4 Methodology . . . 135
5.1.5 Experiments . . . 149
5.1.6 Summary . . . 153
5.2 Dynamic Inter-Block Scheduling . . . 153
5.2.1 Introduction . . . 154
5.2.2 Motivating Example . . . 154
5.2.3 Background . . . 157
5.2.4 Methodology . . . 158
5.2.5 Experiments . . . 169
5.2.6 Summary . . . 172

6 Conclusion **174**

6.1 Outlook . . . 176
6.2 Summary . . . 181

Bibliography **181**

List of Tables

2.1 A summary of benchmark sets used in the rest of the thesis. 32

3.1 Elastic components for dynamically scheduled HLS. 49

3.2 Evaluation of design quality of DASS over eleven benchmarks. 64

4.1 Comparison among our approach with other approaches. 80

4.2 Reference for the scheduling approach that needs to be taken for a loop. 90

4.3 Static islands found over a set of benchmarks. 91

4.4 Comparison between the original wrapper and our new wrapper. 97

4.5 Evaluation of automatically determining static islands on a set of benchmarks. 101

4.6 Component Formalisation of HLS Hardware. 112

4.7 Evaluation of automatically inferring IIs for static islands on a set of benchmarks. 127

4.8 Synthesis time comparison between exhaustive search and our approach. 128

5.1 Performance evaluation for C-slow pipelining. 151

5.2 Evaluation of dynamic C-slow pipelining on a set of benchmarks. 152

5.3 Evaluation of dynamic inter-block scheduling on two benchmark sets. 171

List of Figures

1.1 DASS HLS tool combines the best of two worlds. 3

2.1 An example of FPGA architecture. 10

2.2 An example of FPGA compilation flow. 11

2.3 A typical HLS tool flow. 16

2.4 Schedules of a program using different scheduling approaches. 23

3.1 An example of a program and its execution order of BBs and instructions. . . . 35

3.2 Examples of basic block schedules. 43

3.3 Motivating example for DASS. 46

3.4 Hardware generated from dynamic and static schedules for the example in Figure 3.3. 47

3.5 Dynamic and static schedules for the example in Figure 3.3. 48

3.6 The static island `g` is wrapped with additional control circuitry for interfacing to the dynamically scheduled circuit. 52

3.7 An example of the netlist of all memory accesses for dynamically scheduled hardware. 57

3.8 Shared memory architecture in the DASS hardware. 59

3.9 With user-specified constraints in pragmas, our tool automatically generates a combined dynamically and statically scheduled circuit. 60

3.10 The overall effects of our approach over the eleven benchmarks from Table 3.2. . 63

3.11 LUT usage of different scheduling approaches over the performance for the example from Figure 3.3. 67

3.12 Rate analysis for the motivating example. 70

3.13 Design quality of different scheduling approaches for `bubbleSort` benchmark. . . 73

4.1 A sketch of derectangularising a static island. 76

4.2 Our work integrated into the DASS tool flow. The steps numbered ① to ⑤ are contributions of this section. Section 4.1.4 explains the details. 77

4.3 Motivating example for finding static islands. 78

4.4 Data flow graph from the example in Figure 4.3. 79

4.5 Area and delay of designs for all possible selection of static islands. 81

4.6 A loop example that conditionally updates a variable `s` based on the value of array element `A[i]`. 87

4.7 The loss factor of the loop in Figure 4.6 increases with the probability of `if` condition being true. 89

4.8 An example of a wrapper with offset constraints. 95

4.9 Comparison with the manual DASS designs in [1]. 100

4.10 Comparison with the designs with the best performance. 102

4.11 Motivating example for inferring the optimal II of a static island. 105

4.12 The optimal II of function `ss_func` depends on both the probability of `cond` being true and the dependence distance of `d`. 106

4.13 An example of a Petri net. 108

4.14 A memory controller (MC) is used for balancing the memory bandwidth, and a load-store queue (LSQ) is used for dependence control. 116

4.15 Modelling of memory controllers and load-store queues. 117

4.16 The hardware is modelled by directly stitching up the component models. 118

4.17 Our work of Petri net modelling and analysis integrated into the open-sourced DASS HLS flow. 124

4.18 Area-delay product (ADP) difference between our work and the optimal designs for `vecTrans` over a set of input distributions. 125

5.1 An example of a dynamically pipelined loop starting iterations in order. 131

5.2 Motivating example for dynamic C-slow pipelining. 132

5.3 Schedules for the example in Figure 5.2. 133

5.4 An example of 3-slowing a loop. 134

5.5 Boogie program generated for the example in Figure 5.3. 139

5.6 An example where C-slow pipelining does not improve throughput. 141

5.7 Exploring C for C-slow pipelining from IIs. 143

5.8 Speedup by varying C for the example in Figure 5.3. 145

5.9 Hardware transformation of C-slow pipelining. 147

5.10 Our work integrated into Dynamatic. Our contributions are highlighted in bold blue text. 149

5.11 Motivating example for inter-block scheduling. 154

5.12 Schedules generated by the example in Figure 5.11. 155

5.13 Hardware transformation of the motivating example in Fig. 5.12. 156

5.14 A Boogie program generated for the example in Figure 5.12. 161

5.15 Parallelising BBs that are not completely independent. 163

5.16 An example of parallelising BB starting schedule by CFG transformation. 166

5.17 An example of simplifying data flow for live variables. 167

5.18 Proposed tool flow for dynamic inter-block scheduling. 169

5.19 Speedup, compared to original Dynamatic, as more subgraphs in the CFG are parallelised. 170

6.1 MLIR lowering process in CIRCT HLS [2]. 177

Chapter 1

Introduction

In 2020, 4-6% of global electricity use was by data centres and communication networks [3], and this number is likely to increase due to users' increasing demand for computation. In order to meet the needs for high performance and energy efficiency, custom computing is proposed which represents computation hardware dedicated to a specific application or domain. Such hardware achieves high energy efficiency at the same performance by efficiently exploiting parallelism for dedicated use. This kind of hardware is different from general-purpose processors, such as Central Processing Units (CPUs), which can execute arbitrary software programs.

In custom computing, various hardware architectures are proposed, spanning the trade-off between energy efficiency and flexibility. First, application-specific instruction set processors (ASIPs) are processors that can execute software programs containing a set of pre-defined instructions dedicated for a specific application [4]. Second, field-programmable gate arrays (FPGAs) are integrated circuits based around a matrix of configurable logic blocks connected via programmable switches, enabling people to build custom hardware at the bit level. Finally, application-specific integrated circuits (ASICs) are fully customised hardware dedicated to a single application.

The algorithms in new domains such as machine learning [5] and computer vision [6] are moving fast. General processors and ASIPs may not be able to efficiently exploit the parallelism of a new algorithm on their existing architecture. There is a trend towards offloading flexibility in

software to hardware for more opportunities of parallelism [7]. This makes FPGA and ASIC implementations more valuable for custom computing, since they can be customised down to the bit level. However, traditional ASICs cannot keep up with the pace. The development time for ASIC implementation can be several years, dominated by the design and verification process. Compared to ASICs, the development time for FPGA implementation is significantly less, thanks to the reconfigurability of FPGAs. Users can change their designs simply by updating the configuration of logic blocks and switches, like software programs. FPGAs today can achieve better energy efficiency than ASIPs with comparable performance and require significantly less design time than ASICs. In 2019, Intel claimed that both their field-programmable gate array (FPGA) products, Arria 10 and Stratix 10, can offer 3-10× better energy efficiency relative to NVIDIA Titan X GPU [8]. Today FPGAs are widely used in industry, such as Microsoft Project Catapult [9] and Amazon EC2 F1 instances [10].

Compared to software implementation in software programming languages, such as C/C++, to execute on general-purpose processors and ASIPs, FPGA and ASIC implementations are specified in register transfer level (RTL) descriptions, such as Verilog/VHDL [11]. There are huge differences between the semantics of software programming languages and RTL descriptions. To use such FPGA devices, users must know such description languages and be familiar with detailed digital design at a low abstraction level. This restricts the access to FPGA devices by users without a hardware background, such as many software engineers, even though FPGAs can achieve better performance and energy efficiency.

Aiming to bring the benefits of custom hardware to software engineers, high-level synthesis (HLS) allows the use of a familiar language, such as C/C++, and automatically translates a program into a hardware description. This process can significantly reduce the design effort and time compared to manual RTL implementations. Various HLS tools have been developed in both academia and industry, such as Bambu from the Politecnico di Milano [12], Dynamatic from EPFL [13], Xilinx Vivado HLS [14], Intel HLS Compiler [15], Cadence Stratus HLS [16] and Siemens Catapult HLS [17].

One of the most important tasks for an HLS tool is scheduling: allocating operations to clock

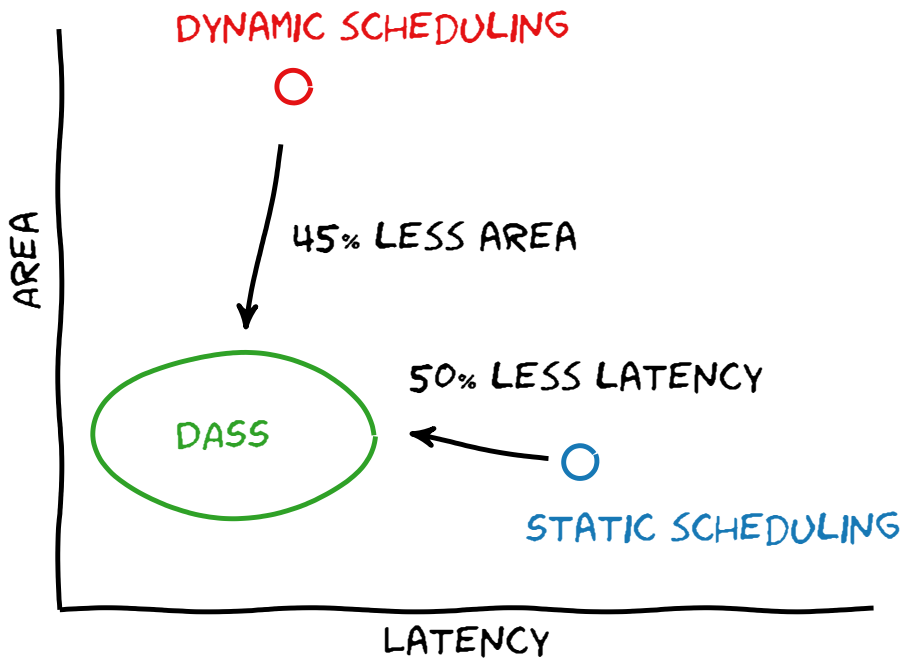


Figure 1.1: DASS HLS tool combines the best of two worlds.

cycles. Scheduling decisions can be made either during the synthesis process (static scheduling) or at run-time (dynamic scheduling).

The advantage of static scheduling is that since the hardware is not yet online, the scheduler has an abundance of time available to make good decisions. It can seek operations that can be performed simultaneously, thereby reducing the latency of the computation. It can also adjust the start times of operations so that resources can be shared between them, thereby reducing the area of the final hardware. However, a static scheduler must make conservative decisions about which control-flow paths will be taken, or how long variable-latency operations will take, because this information is not available until run-time.

Dynamic scheduling, on the other hand, can take advantage of this run-time information. Dynamically scheduled hardware consists of various components that communicate with each other using handshaking signals. This means that operations are carried out as soon as the inputs are valid. In the presence of variable-latency operations, a dynamically scheduled circuit can achieve better performance than a statically scheduled one in terms of clock cycles. However, these handshaking signals may also cause a longer critical path, resulting in a lower operating frequency. In addition, because scheduling decisions are not made until run-time, it is difficult

to enable resource sharing. Because of this, and also because of the overhead of the handshaking circuitry, a dynamically scheduled circuit usually consumes more area than a statically scheduled one.

This thesis tackles the challenges of finding a better scheduling solution by combining the best of both worlds, static scheduling and dynamic scheduling. These challenges include:

1. How to implement an efficient HLS tool that correctly generates efficient dynamically and statically scheduled hardware?

2. How to automatically determine and optimise the code regions that are amenable for static scheduling?

3. How to automatically determine and optimise the code regions that are amenable for dynamic scheduling?

1.1 Research Contributions

This thesis makes the following main contributions:

Our main contributions in Chapter 3 include a formalisation of scheduling in HLS and an implementation of the proposed efficient approach for scheduling. First, existing scheduling formulations either support dependence analysis of static scheduling or dynamic scheduling only. Our proposed formulation describes both static and dynamic scheduling. Our dependence model indicates the dependence constraints to be solved when a scheduling technique is applied, which could either be static or dynamic. Second, we implement an HLS tool named DASS, *i.e.* Dynamic And Static Scheduling, based on the formulation. DASS supports statically scheduling part of the program, while keeping the main program dynamically scheduled. Each statically scheduled region is named a static island. This potentially leads to a better performance in wall clock time because it can achieve better throughput by dynamic scheduling and better clock frequency by static scheduling.

Our main contributions in Chapter 4 include the automation of the scheduling process in the DASS HLS tool. DASS in Chapter 3 requires manual effort for producing hardware designs with both high performance and area efficiency. We use static analysis for automating such steps. First, we propose a heuristic-driven approach to identify code regions that are amenable to static scheduling, and efficiently synthesize them as static islands. Second, we propose a probabilistic approach to estimate the dynamic hardware behaviour at compile time, and automatically balance the throughput between static islands and dynamically scheduled hardware for efficient resource utilisation. These techniques get rid of human efforts in the scheduling flow and efficiently produce hardware designs that have quality close to the manually implemented design by experts.

Our main contribution in Chapter 5 includes advanced dynamic scheduling techniques using static analysis. Existing dynamic scheduling techniques use minimal static analysis for minimising compile time while achieving high performance. Such approaches assume the worst case of dependence and try to capture all of them at run time. We show how to use static analysis to prove the absence of certain dependences in dynamic loops, such as loops with variable loop bounds, and enable more hardware optimisation. First, we propose a technique that automatically identifies the minimum dependence distance of an outer loop in a loop nest for achieving C-slow pipelining. Second, we propose a technique that automatically proves the absence of dependence between sequential loops for achieving inter-block scheduling. These techniques efficiently optimise the hardware design for significantly higher performance at negligible area cost.

1.2 Thesis Outline

The rest of the thesis is organised as follows. Chapter 2 provides the necessary background for the thesis. Section 2.1 introduces FPGAs and their design flow. Section 2.2 presents the background of HLS and its compilation flow. Section 2.3 looks into the scheduling process in HLS and explains different approaches. Section 2.4 introduces and compares a few static

analysis techniques that are useful for optimising the HLS flow. Section 2.5 introduces the benchmarks we used for evaluating the proposed HLS optimisations in this thesis.

Chapter 3 includes two sections. First, Section 3.1 demonstrates our scheduling formulation for both static scheduling and dynamic scheduling in HLS. We also demonstrate a set of possible relaxations for these two models, which could lead to better performance. Second, Section 3.2 demonstrates our implementation of the DASS HLS tool. We describe how our proposal overcomes the challenges related to the integration of static islands and shared memory, and evaluate the effectiveness of DASS on a set of benchmarks and compare the results with the corresponding statically scheduled-only circuits and dynamically scheduled-only circuits.

Chapter 4 includes two sections. First, Section 4.1 demonstrates how to efficiently determine good static islands. We present our wrapper design for static islands and the proposed memory architecture for shared memory, as well as the DASS tool flow, and evaluate the effectiveness of our tool on a set of benchmarks. Second, Section 4.2 demonstrates how to efficiently optimise static islands. We present our Petri net model for statically analysing the dynamic hardware behaviour of a circuit produced by dynamic and static scheduling, and evaluate the effectiveness of our approach on a set of benchmarks.

Chapter 5 includes two sections. First, Section 5.1 shows how to achieve dynamic C-slow pipelining with the help of static analysis. We show how to use Microsoft Boogie to automatically determine the absence of dependence between outer loop iterations and how to efficiently achieve C-slow pipelining by hardware transformation, and evaluate the effectiveness of our approach on a set of benchmarks. Second, Section 5.2 shows how to achieve dynamic inter-block scheduling with the help of static analysis. We show how to use Microsoft Boogie to automatically determine the absence of dependence between sequential loops and how to efficiently realise simultaneous execution of independent sequential loops in hardware. We then evaluate the effectiveness of our approach on a set of benchmarks.

1.3 Statement of Originality

This thesis is my own work, and all related works are appropriately referenced. The original contributions made in this thesis have been published in the following peer-reviewed conference papers and journal articles:

1. **Jianyi Cheng**, Lana Josipović, George A. Constantinides and John Wickerson, "Parallelising Control Flow in Dynamic-Scheduling High-Level Synthesis", *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2022.

2. **Jianyi Cheng**, Lana Josipović, George A. Constantinides and John Wickerson, "Dynamic Inter-Block Scheduling for HLS", in *IEEE International Symposium on Field-Programmable Logic and Applications (FPL)*, 2022.

3. **Jianyi Cheng**, John Wickerson and George A. Constantinides, "Dynamic C-Slowing Pipelining for HLS", in *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2022.

4. **Jianyi Cheng**, John Wickerson and George A. Constantinides, "Finding and Finessing Static Islands in Dynamically Scheduled Circuits", in *the ACM International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2022.

5. **Jianyi Cheng**, John Wickerson and George A. Constantinides, "Exploiting the Correlation between Dependence Distance and Latency in Loop Pipelining", in *IEEE International Symposium on Field-Programmable Logic and Applications (FPL)*, 2021.

6. **Jianyi Cheng**, John Wickerson and George A. Constantinides, "Probabilistic Scheduling in High-level Synthesis", in *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2021.

7. **Jianyi Cheng**, Lana Josipović, George A. Constantinides, Paolo Ienne and John Wickerson, "DASS: Combining Dynamic & Static Scheduling in High-level Synthesis", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (IEEE TCAD)*, 2021.

8. **Jianyi Cheng**, Lana Josipović, George A. Constantinides, Paolo Ienne and John Wickerson, "Combining Dynamic & Static Scheduling in High-level Synthesis", in *the ACM International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2020.

Chapter 3 covers the research published at TRETS 2022, FPL 2021, TCAD 2021 and FPGA 2020; Chapter 4 covers the research published at FPGA 2022 and FCCM 2021; and Chapter 5 covers the research published at FPL 2022 and FCCM 2022.

Chapter 2

Background

In this chapter, we will first introduce Field Programmable Gate Arrays and their design flow. Second, we will present the background of high-level synthesis (HLS) and its compilation flow. Next, we look into the scheduling process in HLS and explain different approaches. We then introduce a few static analysis techniques that are useful for optimising the HLS flow. Finally, we introduce a few sets of benchmarks that are used for evaluating our work in the thesis.

2.1 Field Programmable Gate Array

Field Programmable Gate Arrays (FPGAs) are integrated circuits based around a matrix of configurable logic blocks connected via programmable switches, enabling people to build custom hardware [18, 19, 20]. In an FPGA, each configurable logic block contains several logic cells, where the number of logic cells depends on the architecture of FPGAs.

For most FPGAs, a reconfigurable logic cell contains a look-up table (LUT), a multiplexer and a D flip-flop [21]. The LUT is usually a 4-input LUT. The functionality of the logic cell is configured by the inputs to the LUT, which could either be an arithmetic operation or a logic operation. The result could either be asynchronous or synchronous. This is configured by selecting between the input and the output of the D flip-flop using the multiplexer. An

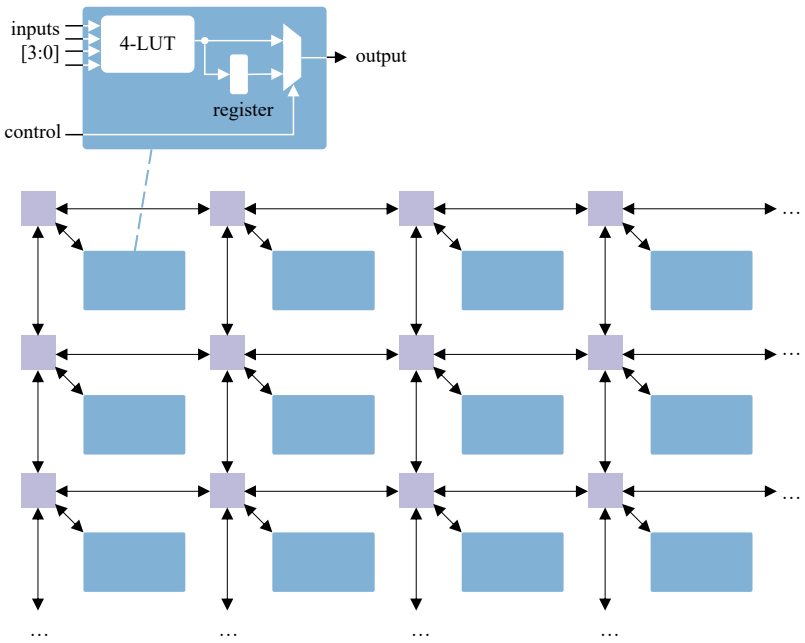
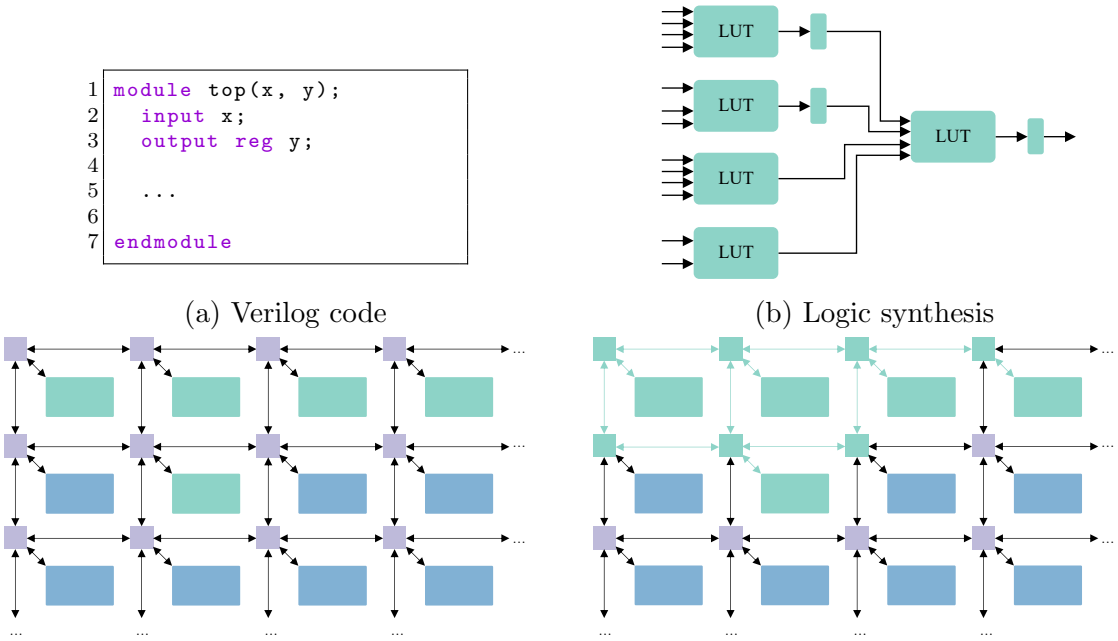


Figure 2.1: An example of FPGA architecture.

example of a reconfigurable logic cell is shown at the top of Figure 2.1. A modern FPGA also has digital signal processors (DSPs) [22, 23] for high-performance arithmetic operations and on-chip random-access memory blocks (BRAMs) [24] for efficient memory storage. These modules and logic cells are mapped as arrays in an FPGA and connected through a set of programmable switches. Each switch is made of several multiplexers, enabling connections between any two connected modules. The bottom of Figure 2.1 shows an example of FPGA architecture. Each big blue rectangle is a reconfigurable logic cell, and each small purple square is a switch.

When programming with an FPGA, users describe their application as custom hardware in a low-level language such as Verilog [11] or VHDL [25]. Figure 2.2a shows an example of Verilog code. The description is then parsed by the corresponding front end and lowered into a netlist consisting of BRAMs, DSPs and LUTs. This process is called logic synthesis [26, 27], which maps target-independent specifications of a hardware design onto a given architecture. An example of the netlist after logic synthesis is shown in Figure 2.2b. Then the netlist is placed onto exact components in the FPGA and routed through the programmable switches while satisfying all the timing and resource constraints specified by the user. This process is called placing and routing [28], as illustrated in Figure 2.2c and Figure 2.2d. The output after the placing and routing process is then translated to a bitstream for the target FPGA. The



(a) Verilog code

(b) Logic synthesis

(c) Placement

(d) Routing

Figure 2.2: An example of FPGA compilation flow.

FPGA can perform the same functionality as the custom hardware design when the bitstream is downloaded [29].

Such reconfigurability of an FPGA enables users to customise their hardware design in a programmable form, which avoids the complex taping-out process in ASIC design flows and accelerates the prototyping process. Still, there are two challenges of programming with FPGAs. First, the hardware description languages used for programming FPGAs are usually precise to bit level but inefficient. For example, a simple design may require many lines of code to be correctly specified. Writing all the code manually takes a lot of effort. In addition, hardware description languages have huge differences from software programming languages and require users to have low-level hardware knowledge. This makes it difficult for people without hardware background, such as software engineers, to build their custom hardware.

2.2 High-Level Synthesis

High-level synthesis (HLS) is proposed to overcome these two challenges. HLS is a process that automatically translates high-level software programs to low-level hardware descriptions. In the

hardware community, HLS tools may have two classes of users. First, hardware engineers use HLS tools as a code generator for efficiently implementing their hardware using a small number of lines of code. They manually add customised hardware constraints in an efficient form, such as pragmas, into the input program for guiding the HLS tool to generate efficient hardware. The quality of the final hardware is usually as good as manually written RTL designs. This solves the first challenge. However, the HLS code into the HLS tool still requires significant human effort for these manual optimisations, and the HLS code is hard to read as the code representation is more similar to RTL descriptions than a general software program.

Second, software engineers use HLS tools for efficiently implementing efficient hardware. Because most software engineers do not have as much hardware knowledge as hardware engineers, manual hardware optimisation is challenging for them. Instead, HLS tools play an important role in the hardware optimisation process. An HLS tool analyses the input program and automatically explores possible optimisation opportunities. This usually results in a set of good design constraints. Such a design flow requires less effort from users, and the input program is readable and similar to any software program. This reduces the second challenge. However, the performance or area of the final hardware may not be as good as the manually written RTL designs by experts. The scope of this thesis targets the latter case, where our work aims to take an arbitrary software program and automatically generate an efficient hardware design.

2.2.1 HLS Tool Framework

This section introduces a few compiler frameworks that are commonly used for implementing HLS tools. First, we introduce the most commonly used compiler framework named LLVM. Second, we introduce a newer compiler framework used by recent HLS tools named multi-level intermediate representation (MLIR). Finally, we also introduce other frameworks that are used for implementing HLS tools.

LLVM-based HLS Tools

LLVM is one of the most well-known compilers and toolchain frameworks that allows tool designers to design their front end for any programming language and their back end for any instruction set architecture [30]. In the HLS world, the LLVM framework has also been widely used for implementing HLS tools, including Vivado HLS [14], Intel HLS compiler [15], LegUp [31] and Dynamatic [13].

LLVM has its own low-level intermediate representation (IR) named LLVM IR. This IR is a target-independent assembly language which can take an infinite number of function local registers [30]. An LLVM-based compiler takes the input program and translates it to LLVM IR for program analysis and transformation. The transformed LLVM IR is then translated to a target-specific language in the back end. LLVM IR has a few data structures:

Module: A module represents a single set of code that is to be processed together, which is the top level of the design.

Function: A module contains a set of functions.

Basic Block: A function contains a set of connected basic blocks (BBs). A basic block is a list of straight-line code followed by a branch instruction. The connections between basic blocks represent the control flow of the function.

Instruction: A basic block contains a list of instructions. An instruction represents one of the specific operations, such as arithmetic, logic, and memory operations. The connections between instructions represent the data flow of the basic block.

When implementing an HLS tool in LLVM, existing front-end tools can be directly reused, which translates an input program to LLVM IR. For typical HLS tools that take C programs as input, a front-end tool named Clang [32] is used to emit LLVM IR from C programs. However, for the back end, no RTL target is available in the LLVM upstream because the LLVM framework was proposed mostly for processor targets. A dedicated RTL target needs to be implemented

in the back end for RTL code generation. For instance, LegUp [31] contains a Verilog target in its back end, and Dynamatic has a VHDL target in its back end.

One of the biggest advantages of building HLS tools using the LLVM framework is that the tool designers can reuse the existing analysis and optimisation passes for software programs in the LLVM upstream. This significantly lifts the workload compared to building a tool from scratch. As one of the most widely used compiler frameworks, the LLVM framework contains several passes that can be directly used for HLS optimisation, such as constant propagation, common subexpression elimination (CSE) and if-conversion. Details of these passes are explained in Section 2.2.3.

Multi-Level Intermediate Representation-based HLS Tools

LLVM today has been widely used in industry, such as Apple's Swift compiler [33], and Intel's compiler [34]. However, it only has one intermediate representation and might not be efficient for compiling all kinds of programming languages. For instance, canonicalised loop information in C/C++ cannot be well maintained by the default structs in LLVM IR. Users need to annotate it informally in the metadata before it gets used in the later stages of the compilation flow, otherwise it may not get recovered after the IR gets transformed by some passes in the early stages. Lowering a program in an arbitrary programming language to LLVM IR could be a significantly lossy process, where the lost information related to the features where the programming language might miss optimisation opportunities.

In order to minimise the information loss in each step during the compilation process, multi-level intermediate representation (MLIR) is proposed. MLIR contains multiple forms of IR, named dialects, including an LLVM IR dialect for representing LLVM IR. The MLIR framework reduces the information loss during transformation by progressively lowering an IR to a low-level IR. Such a framework enables analysis and optimisation to be carried out at different abstractions. Also, MLIR allows different parts of a program to be represented in different dialects, and each part can be optimised using dedicated passes. There is a trend of using MLIR for implementing compilers instead of LLVM. An example of an MLIR-based HLS tool

will be introduced in Chapter 6.

Another advantage of implementing an HLS tool using MLIR is that a tool designer can customise their own abstraction in the same framework as existing MLIR dialects for dedicated analysis and transformation. For instance, an HLS tool named ScaleHLS implements an `HLS` dialect higher than `affine` dialect for design space exploration of HLS pragmas in MLIR [35]. An HLS tool named HeteroCL implements a dialect named `HCL`, an abbreviation for HeteroCL, also higher than the `affine` dialect for domain-specific code optimisations [36].

Other Frameworks

Most HLS tools accept C/C++ programs as C/C++ is one of the most popular programming languages in the world. These tools usually use the LLVM or MLIR frameworks. However, there are HLS tools that accept other programming languages and are implemented in frameworks from software frameworks for those program languages. For instance, an HLS tool accepting Haskell programs [37] is implemented in Cλash [38]; an HLS tool accepting Python programs [39] is implemented in Python [40]; and an HLS tool accepting Julia programs [41] is implemented in Julia [42]. There are also HLS tools that accept C-like languages, such as OpenCL [15, 16] and SystemC [14, 15], implemented in their own frameworks.

Apart from accepting programs in existing software programming languages, there are HLS tools accepting customised input languages. These languages are usually tailored for specific domains, also known as domain-specific languages (DSL). A DSL restricts the input program to have a pre-defined set of features, which significantly simplifies program analysis in the tool flow. More aggressive hardware optimisation techniques can be applied to a DSL program compared to optimising a general program. For instance, HeteroCL [36] proposes an HLS-specific DSL, modified from a software DSL named Halide [43], for hardware synthesis of regular kernels, such as stencil computation. FPGAConvNet [44] proposes a graph-level DSL, modified from ONNX graph [45], for hardware synthesis of convolutional neural network accelerators.

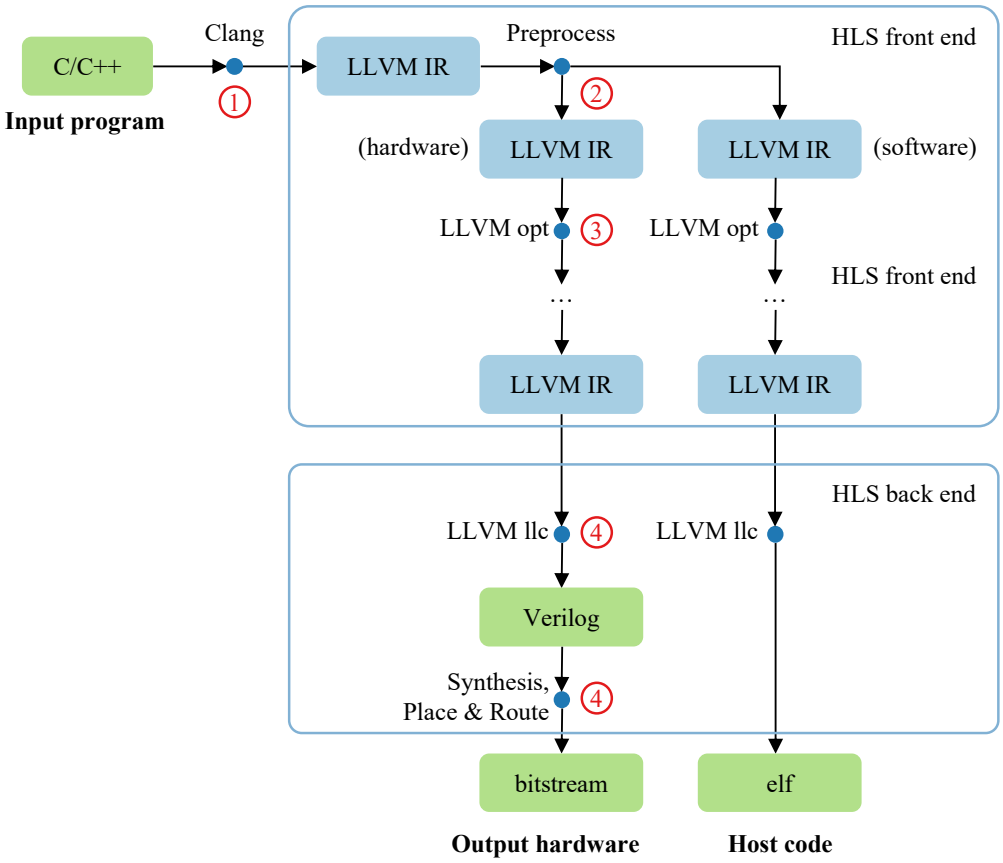


Figure 2.3: A typical HLS tool flow.

2.2.2 HLS Tool Flow

We now illustrate the methodology of HLS design flow. A traditional HLS tool flow is illustrated in Figure 2.3, which consists of the following passes:

① The input program is parsed and translated into a software IR, such as LLVM IR, via a front end, such as C compiled by Clang [32].

② Each hardware function is considered as a hardware instance and transformed into hardware IR. The software functions remain the representation in software IR as host code.

③ The hardware functions are transformed multiple times by a series of optimisation passes: some hardware specific, such as word-length minimisation, and others generic, such as dead-code elimination. This includes the scheduling process, which maps the start times of operations into clock cycles. The details of scheduling will be explained later in this chapter. For the software functions, an additional transformation is made to implement

the interface between software and hardware functions as appropriate hardware function calls.

④ Each transformed hardware function is then turned into an RTL description of a circuit. Interconnect logic and memory interfaces are generated to connect each of the circuits to the host system and instantiate them the appropriate number of times.

⑤ The software host code is compiled, and an FPGA hardware bitstream is generated by synthesising the RTL code using FPGA vendor tools.

Stages ①, ② and ③ are often referred to as the front end and stages ④, and ⑤ are referred to as the backend. Different HLS tools have their dedicated features along these stages. For instance, LegUp [31] supports hardware synthesis from multi-threaded code; and Dynamatic [13] supports the synthesis of dynamic scheduled hardware.

2.2.3 HLS Passes for Hardware Optimisation

HLS tools apply various optimisation passes onto the input program for producing high-performance and area-efficient hardware. These optimisation passes can be divided into two groups, front end and back end.

The optimisation passes in the front end are mainly software-level optimisations using source-to-source optimisations. Most of these passes already exist in the software tool flow and can be directly used. Here we give a few examples of reusing existing software passes for hardware optimisation.

Constant propagation substitutes the values of known constants in expressions. In software compilation, this process saves execution latency; and in HLS, it also saves hardware resources by replacing constant-input hardware datapaths implemented using logic cells with constants implemented using wires.

Common subexpression elimination (CSE) replaces an instance of identical expression to an existing one with a single variable holding the computed value. In software compilation, this process saves execution latency by avoiding repeated execution; and in HLS, it also saves hardware resources by replacing duplicated datapaths with shared wires.

Dead Code Elimination (DCE) removes code that does not affect the program results. In software compilation, this process saves execution latency by avoiding irrelevant execution; and in HLS, it also removes unnecessary hardware resources.

Promoting memory to register(mem2reg) promotes a single scalar variable from memory to register. In software compilation, this process saves execution latency by avoiding some load and store operations; In HLS, it also uses registers or wires to save BRAM blocks.

Loop unrolling optimises the execution latency by increasing the number of instructions in an iteration. In software compilation, this process saves execution latency by having fewer branches; but in HLS, it duplicates hardware instances of instructions for significantly more parallelism.

Array Partitioning partitions an array into multiple sub-arrays. In software compilation, this process enables efficient data storage by storing sub-arrays in different memory systems; but in HLS, it leads to better performance by enabling higher memory bandwidth when accessing two sub-arrays in different BRAM blocks in parallel.

The optimisations in the back end require a hardware abstraction beyond sequential software programs and need to be carried out in a hardware abstraction. Here we give a few examples of essential back-end optimisations for HLS.

Scheduling exploits parallelism in a spatial form for better latecny. Details of such a process will be explained in section 2.3. Scheduling also includes re-timing that enables multiple dependent operations to be executed in a single clock cycle. Scheduling is one of the most important processes that can significantly improve performance.

Binding uses a single hardware instance for computing two instructions at different times. This is also known as resource sharing. This achieves efficient hardware resource utilisation with minimal performance overhead.

Pipelining exploits parallelism between iterations of the same instance for better throughput. This is usually done at either loop level or functional level. Pipelining is also one of the most important processes that can significantly improve performance.

Buffering improves performance by matching the performance between two connected instances.

IP selection optimises performance or area efficiency by selecting suitable hardware components based on the given constraints.

In this thesis, we focus on the scheduling and pipelining process in HLS. We show how to optimise existing scheduling approaches for achieving better performance. Scheduling and pipelining are usually carried out together, here we use the term 'scheduling' for the whole process.

2.3 Scheduling

Scheduling is a process that maps the execution order of operations. When scheduling a program, the program is usually translated into a control/data flow graph (CDFG) for program analysis [7]. A CDFG is a two-level directed graph consisting of a number of vertices connected by edges. At the top level, the graph is represented as a control-flow graph (CFG), where each vertex corresponds to a BB in the transformed IR, while edges represent the control flow. At a lower level, a vertex corresponding to a BB is itself a data-flow graph (DFG) that contains a number of sub-vertices and sub-edges. Each sub-vertex represents an operation in the BB and each sub-edge indicates a data dependency.

In software compilation, the hardware architecture is fixed and barely reconfigurable, such as

the number of cores and threads. Scheduling in software tries to determine a correct execution order with maximal parallelism and minimal pipeline stalls given a fixed hardware architecture.

Scheduling in HLS tries to determine an execution schedule with a minimal latency in clock cycles and an allocation plan for available hardware resources. It defines the hardware architecture from the input software program, which potentially exploits more parallelism in custom hardware design. Scheduling in HLS maps the start times of the operations into clock cycles, instead of changing the execution order only. The hardware resources are more flexible to program for efficiently executing various operations. For instance, an HLS program can either be parallelised in 8 threads computing complex tasks or 1024 threads computing simple tasks.

Traditionally, there are two approaches for scheduling an HLS program, static scheduling and dynamic scheduling.

2.3.1 Static Scheduling

Static scheduling is performed at compile time. In HLS, the static scheduler determines the start and end clock cycles of each operation in the CDFG, under which the control flow, data dependencies, and constraints on latency and hardware resources, are all satisfied. One of the most common static scheduling techniques, used by Vivado HLS [14] and LegUp [31], expresses a CDFG schedule as a solution to a system of difference constraints (SDC) [46]. Specifically, it formulates scheduling as a linear programming (LP) problem, where the data dependencies and resource constraints are represented as inequalities. By changing these constraints, various scheduling objectives can be customised for the user's timing requirements. Figure 2.4 shows schedules of a program using different approaches. Detailed formulation of static scheduling is explained in Chapter 3.

Besides achieving high performance, static scheduling also takes resource allocation into account, such as modulo scheduling for loop pipelining [47]. It aims to satisfy the given time constraints with the minimum possible hardware resources or achieve the best possible performance under the given hardware resource requirements. If the hardware resource constraints

are not specified, the binder automatically shares some hardware resources among the operations that are not executed in parallel. This maintains the performance but results in a smaller area. In addition, typical HLS tools like Vivado HLS allow users to specify resource constraints via pragmas. In this case, the binder statically fits all operations into a given number of operators or functions based on the given schedule. This may slow down execution if hardware resources are limited.

A statically scheduled circuit is generated as a single finite state machine (FSM) with a pre-determined schedule and datapath. The FSM controls the computation of all the operations following the pre-determined schedule.

In summary, static scheduling results in efficient hardware utilisation by relying on the knowledge of the start times of the operations to share resources while preserving high performance. However, when the source code has variable-latency operations or statically indeterminable data and control dependencies, static scheduling conservatively schedules the start times of certain operations to account for the worst-case timing scenario, hence limiting the overall throughput and achievable performance.

2.3.2 Dynamic Scheduling

Dynamic scheduling is a process that schedules operations at run-time. It overcomes the conservatism of traditional static scheduling to achieve higher throughput in irregular and control-dominated applications, as we saw in Figure 2.4b. Similarly, dynamic scheduling can handle applications with memory accesses which cannot be determined at compile time. For instance, given a statement like `x[h1[i]] = g(x[h2[i]])`, the next read of `x` can begin as soon as it has been determined that there is no read-after-write dependency with any pending store from any of the previous loop iterations, *i.e.* `h2` is not equal to any prior/pending store address `h1`. A dynamically scheduled circuit will allow the next operation to begin as soon as this inequality has been determined; otherwise, it will appropriately stall the conflicting memory access.

Dynamic scheduling for asynchronous circuits has been well-studied in the past few decades. Ini-

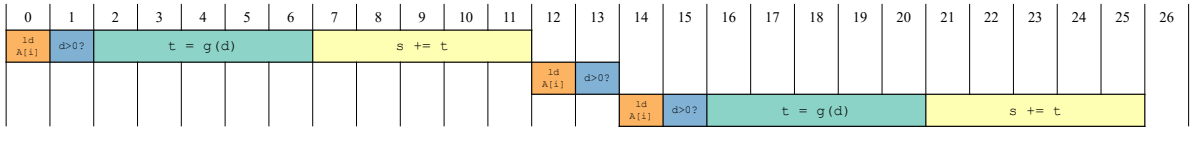
tial tool flows perform syntax-directed translation from high-level HDL [48, 49, 50]. Yoneda *et al.* propose a tool flow that produces asynchronous circuits from Petri nets [51]. Nielsen *et al.* propose a tool flow that enables HLS of asynchronous hardware from programs in a language named Balsa [52], which was later extended to support programs in a language named HASTE [53].

Venkataramani *et al.* [54] proposed a framework that automatically transforms a C program into an asynchronous circuit. They implement each node in a DFG of the design in an intermediate representation called Pegasus [55] into a pipeline stage. Each node represents a hardware component in the netlist, containing its own controlling trigger. Li *et al.* [56] propose an HLS tool named Fluid that can handle complex CFGs more efficiently.

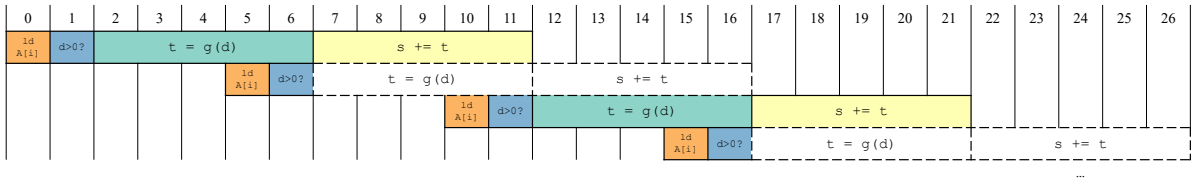
Dynamically scheduled synchronous hardware synthesis from a high-level language was initially proposed by Luk and Page [57]. They propose a framework for automatically mapping a program in occam into a synchronous hardware netlist. This work was later extended to a commercial language named Handel-C [58]. However, it still required the designer to manually design and implement hardware optimisation such as pipelining and parallelism. Recent work [13] proposes a tool flow named Dynamatic that generates synchronous dataflow circuits from C code. It can take arbitrary input code, automatically exploits the parallelism of the hardware and uses handshaking signals for dynamic scheduling to achieve high throughput. In this work, we use Dynamatic to generate dynamically scheduled HLS hardware.

As formalised by Carloni *et al.* [59], dynamic scheduling is typically implemented by dataflow circuits which consist of components communicating using handshake signals. Apart from the common datapath operators, a dynamically scheduled dataflow circuit contains a number of dataflow components used to control the flow of data.

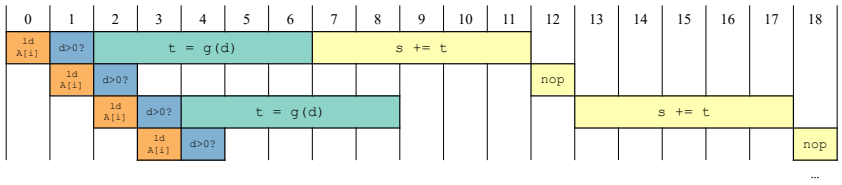
One difficulty for dynamic scheduling is scheduling the memory accesses. In static scheduling, all the memory accesses are scheduled at compile-time, such that there is no memory conflict during the execution. In dynamic scheduling, the untimed memory accesses may affect correctness and performance if the memory arbitration or the memory conflicting accesses are not correctly solved. Hence, dynamically scheduled circuits may use load-store queues (LSQs) [60]



(a) Static schedule (sequential).



(b) Static schedule (pipelined).



(c) Dynamic schedule.

Figure 2.4: Schedules of a program using different scheduling approaches.

to resolve data dependencies and appropriately schedule the arbitrary memory accesses at run-time. However, LSQs cost significant area and add delays to the circuit. To reduce this overhead, the architecture of the LSQ in Dynamatic is optimised by statically identifying the memory aliasing among all the memory accesses [61]. Detailed formulation of dynamic scheduling is explained in Chapter 3.

Figure 2.4a illustrates the schedule of sequentially executing the program. There is no parallelism in the program. Figure 2.4b illustrates the statically pipelined schedule of the same program. The execution of two consecutive iterations is in parallel. The start times of an operation in two consecutive iterations, also known as the initiation interval (II), is a fixed number of clock cycles. Figure 2.4c illustrates the corresponding dynamically pipelined schedule. The II is now a variable changing at run time. A detailed discussion of this example is provided in Chapter 3.

2.3.3 Static and Dynamic Scheduling

Several works have explored extending static scheduling to support dynamic optimisations. Alle *et al.* [62] and Liu *et al.* [63] propose source-to-source transformations to enable multiple schedules selected at run-time after all the required values are known. Tan *et al.* [64] propose an approach named ElasticFlow to optimise pipelining of irregular loop nests that contain dynamically-bound inner loops. Dai *et al.* [65, 66] propose pipeline flushing for high throughput of the pipeline and dynamic hazard detection circuitry for speculation in specific applications.

In comparison, these works enable static and dynamic scheduling by adding dynamic mechanisms in the form of hardware logic to the statically scheduled hardware. These hardware designs are still based on static scheduling and usually work only under stringent constraints. This limits the performance improvements for general cases, such as input-dependent conditions and unpredictable memory accesses. In contrast, this thesis adds optimisations from static scheduling to dynamically scheduled hardware. Both works exploit the efficient design points between static scheduling and dynamic scheduling. Our work improves the area efficiency of the dynamically scheduled hardware and preserves its good performance, while the related works improve the performance of statically scheduled hardware and preserve its good area efficiency.

Carloni [67] describes the theory of how to encapsulate static modules into a latency-insensitive system, and we use a similar integration philosophy. We utilise this approach within our tool, which automates the generation of circuits from high-level code, resulting in the mix of two HLS paradigms in a single synthesis tool. Our designs are all synchronous.

2.4 Static Analysis for HLS

Static analysis is a technique that extracts information from a program without executing it. Such information can be used to optimise the compilation process for better execution. In HLS, static analysis has been used for exploiting more parallelism by identifying the absence of dependences between operations, and saving more area by identifying the absence of concurrently

accessing a certain hardware resource. Here we introduce a few static analysis techniques that have been used in HLS tools.

2.4.1 Polyhedral Techniques

The polyhedral model [68, 69] is a mathematical framework that efficiently describes complex program behaviours. It is usually used for analysing nested loops with affine memory accesses. An affine memory access has an access pattern of the following form:

$$y = ax + b \tag{2.1}$$

y is the accessed array index. x is the dimension, which is usually the iterator of the innermost loop where the memory access is carried out. a and b are known as symbols, which are loop invariants. For a loop nest, its polyhedral model describes each loop as a lattice point. Transformation can be carried out by converting these loops into equivalent expressions, but may have better execution performance.

Polyhedral techniques are popular for memory analysis [70], and have been extended to HLS by the work of Liu *et al.* [71]. Polyhedral techniques exploit parallelism among memory accesses, which results in various HLS optimisations. First, the on-chip memory could be efficiently partitioned to increase its bandwidth for parallel memory accesses [72, 73]. Second, a loop nest can be transformed into a set of parallel loops for better performance [63, 74, 75, 76].

Today, polyhedral analysis has been widely used in HLS tools. However, there are still restrictions as it can only analyse affine memory accesses. Analysis of non-affine memory access remains challenging for HLS optimisation today.

2.4.2 Formal Verification

Formal verification is a technique that uses formal methods of mathematics to prove or disprove the correctness of a system, either in software or hardware, based on a set of formal specifications

or properties of the system. Formal verification has been widely used for verifying software and hardware systems, such as concurrent memory systems [77], compiler designs [78] and microprocessor implementation [79].

The formal verification is usually carried out using satisfiability modulo theory (SMT) [80], which determines whether a mathematical condition is satisfiable. Compared to the Boolean satisfiability problem (SAT) which only accepts Boolean constraints, SMT is more general and supports more constraint types, such as integers and real numbers, and complex constraint constructs, such as lists and bit vectors. An SMT problem can be automatically solved by SMT solvers, such as Z3 [80] and cvc5 [81] for a practical subset of inputs.

There are various verification languages and tools available using different abstractions. z3 [80] is a verification language that describes SMT problems at low level. Dafny [82] and Boogie [83] are verification languages that describe SMT problems at high level. In the hardware verification world, Cadence JasperGold [84] is one of the well-known tools for SystemVerilog assertion verification. It aims to prove the correctness of a hardware design based on the value of the results and the computation time. In this thesis, we use Boogie to describe SMT problems as it has similar structs to the input C programs to our HLS tool.

Boogie is an automatic program verifier from Microsoft Research, built on top of SMT solvers [83]. Boogie uses its own intermediate verification language (IVL) to represent the behaviour of the program being verified. Instead of executing the program, an SMT solver is applied to reason about the program's behaviour, including the values that its variables may take. Encoding of verification as SMT queries is automatically performed by Boogie, hidden from the user. Other works have proposed the automated translation of an original program to Boogie IVL, such as SMACK [85], an automated translator of LLVM-IR code into the equivalent Boogie program. In Chapter 5, we show that we built our own code generation because our Boogie program is specifically designed to solve a particular problem.

The SMT problem is typically NP-hard and is challenging to solve for complex theories. In practice, most software and hardware systems are too complex to be formally verified. There are ongoing research works accelerating the solving process by reduce the complexity of prob-

lem [86]. Recently, there has been an interest in formally verifying small properties of a complex system for exploiting more optimisation opportunities instead of verifying the correctness of the system. Zhou *et al.* [87] propose a SMT-based approach to verify the absence of memory contention in banked memory among parallel kernels. Our prior work proposes a Boogie-based approach for simplifying memory arbitration for multi-threaded hardware [88]. In Chapter 5, we show how to use formal verification to prove the absence of memory dependence for exploiting control flow parallelism.

2.4.3 Probabilistic Analysis

Probabilistic analysis estimates the average behaviours of a system in a mathematical model. A probabilistic analyser assumes a probabilistic distribution of each input to the system, and efficiently explores the distribution of the outputs of the system.

Probabilistic analysis is more suitable for solving optimisation problems as it allows the capture of a large number of possible cases within a very compact representation that can be efficiently explored by existing tools. Probabilistic optimisation has been studied as the result does not affect the correctness but the quality of the system [89, 90]. However, probabilistic optimisation for HLS is under-explored. In Chapter 4, we show how to use probabilistic analysis for optimising dynamically and statically scheduled hardware.

2.4.4 Comparison

Here we introduce three categories of static analysis that potentially help HLS tools for generating a better hardware design. First, polyhedral techniques efficiently explore the parallelism for an input program but only work for affine operations. Polyhedral techniques are useful for programs that consist of regular computation kernels such as constant bounded `for` loops and affine memory operations. Second, formal verification-based approaches can analyse dependence among non-affine operations, but it is restricted by its scalability. These approaches are more suitable for proving small properties in the optimisation stage. Both these two categories

analyse whether a property is always true or false at compile time. This means that they can only capture the worst case of the computation behaviour in the analysis. Finally, probabilistic analysis models the average program behaviour, and estimates the probabilistic distribution of an output or a property. It is sometimes helpful for optimisation by estimating the steady state of a system.

2.5 Benchmarks

Benchmarks are used in practice for evaluating an approach by comparing its output with other approaches. In HLS tool design, there are several benchmark suites that have been widely used, including Polybench [91], CHStone [92], Livermore kernels [93] and Rosetta [94]. These benchmarks tend to be tailored to what HLS tools already comfortably handle. In order to show how our work pushes the limits of HLS, we use a set of benchmarks that have dynamic behaviours and are representative of commonly used code patterns.

We select a number of benchmarks that are amenable for dynamic scheduling based on whether they lack the features mentioned in Section 2.3.2, that is, fully static scheduling is suboptimal. Specifically, these benchmarks all have unpredictable behaviours at run time, such as data-dependent conditions, variable loop bounds, and unpredictable memory accesses. All the benchmarks are made publicly available [95].

Section 3.2 evaluates our work on benchmark set 1, which contains following benchmarks:

- `sparseMatrPow` performs the dot product of two sparse matrices, which skips the operation when the weight is zero.
- `histogram` sums various weights onto the corresponding features but also in a sparse form.
- `filterSum` conditionally sums a number of polynomial results of two arrays. It is a micro-benchmark shown in Figure 3.3.

- `filterSumIf` is similar to `filterSum` but it sums two polynomial expressions based on the value of the difference.
- `getTanh` evaluates the approximated function *tanh(x)* onto an array of integers using the CORDIC algorithm [96] and a polynomial function.
- `getTanh(double)` is similar but uses an array of doubles.
- `BNNKernel` is a small binarised neural network [97].
- `bubbleSort` is a bubble sort algorithm that repeatedly swaps the elements in the one-dimensional array until the sequence is in ascending order.
- `LFK7` is kernel 7 in the Livermore loops [93], a well-known benchmark set for loop kernels.
- `distSum` evaluates the probability of three events in a specific domain by accumulating the three probability density functions (pdfs).
- `getIntersection` measures the volume intersection of polyhedra, used for modelling tumours in biophotonic cancer treatments [98].

`sparseMatrPow` and `histogram` are the sparse form of the corresponding benchmarks from the paper by Josipović *et al.* [13]. `filterSum` and `filterSumIf` are made artificially to demonstrate simple examples. The two `getTanh` benchmarks apply the existing approximation algorithms on sparse data arrays.

Section 4.1 also uses the benchmark set above to compare the results between the designs from our automated flow and the designs from manual optimisation by experts. We also include benchmark set 2, which contains more complex benchmarks, to show how our approach is scalable and can handle large benchmarks:

- `doitgenTriple` is a weighted version of the multi-resolution analysis kernel (MADNESS) [91].
- `correlation` computes the correlation of two matrices.

- `levmarq` is an implementation of the Levenberg-Marquardt algorithm to solve least-squares problems [99].
- `gramSchmidt` is an optimised version of Gram-Schmidt decomposition, which conditionally computes a matrix based on the 2-norm of the rows in a matrix [100].
- `covariance` computes the covariance matrix.
- `syr2k` is a symmetric rank-2k update for two matrices.
- `gemver` is vector multiplication and matrix addition.
- `gesummv` is scalar, vector and matrix multiplication.

Most of them are modified from Polybench. In the first six benchmarks, regular computations are translated into sparse computations to improve performance. The last four benchmarks have complex loop nests. Section 4.1 will also include a motivating example `vecNormTrans`, shown in Figure 4.3.

Section 4.2 needs to show that resource sharing within a static island is beneficial for high area efficiency. Benchmark sets 1 and 2 only show the benefits of dynamic scheduling. We use benchmark set 3 where the static islands have the opportunities for resource sharing, that is, II$>$ 1 can be beneficial:

- `vecTrans` is the motivating example in Section 4.2, shown in Figure 4.11.
- `vecTrans2` is similar to the motivating example, however, the `store` operation is conditional depending on the array data.
- `vecTrans3` is also similar to the motivating example but all the memory accesses are indirectly addressed. The type of array data is floating-point.
- `evalPos` is an evaluation function for a chess engine, which evaluates the given position on the board [101].

- `chaosNCG` is a function for the Naive Czyzewski Generator in the Chaos engine to pull the data from the buffer [102].

The benchmarks above show the benefits of statically scheduling part of the program. Section 5.1 evaluates our work on dynamic C-slow pipelining, which requires benchmarks that have complex control flows.

The evaluation in Section 5.2 also requires complex control flows. Benchmark set 2 is reused. However, the dynamic inter-block scheduling approach is amenable to multiple sequential loops. The applicable benchmarks in benchmark set 2 are still limited. Here we use benchmark set 4, which is a modified version of the LegUp benchmark set by Chen and Anderson [103] for evaluating multi-threaded HLS. The LegUp benchmark set manually specifies the threads using Pthreads [104]. We inlined all the threads to a sequential program. In order to create more opportunities for our optimisation to be applied, we unrolled the outermost loops by a factor of 8. This is the largest factor that still led to the designs fitting our target FPGA. We also partitioned the memory using the block scheme to increase memory bandwidth. The benchmarks in benchmark set 4 are listed as follows:

- `histogram` constructs a histogram from an integer array,
- `matrixadd` sums a float array,
- `matrixmult` multiplies two float matrices,
- `matrixtrans` transposes a single matrix,
- `substring` searches for a pattern in an input string,
- `los` checks for obstacles on a map,
- `fft` performs the fast Fourier transformation,
- `trVecAccum` transforms a triangular matrix,
- `covariance` computes the covariance matrix,

Table 2.1: A summary of benchmark sets used in the rest of the thesis. ✓= applicable. ✗= not applicable.

Sections	1	2	3	4
3.2	✓	✗	✗	✗
4.1	✓	✓	✗	✗
4.2	✗	✗	✓	✗
5.1	✗	✓	✗	✗
5.2	✗	✓	✗	✓

- `syr2k` is a symmetric rank-2k matrix update, and
- `gesummv` is scalar, vector and matrix multiplication.

In summary, the benchmark sets used for the following chapters are shown in Table 2.1. Applicable benchmarks mean that they have the code features that can be transformed by the evaluated approach and cause differences in area or performance compared to the baselines.

2.6 Summary

In this chapter, we introduced the background of FPGAs and their compilation flow. We also demonstrated how HLS makes designing hardware more accessible to a broader range of people. We then introduced how HLS tools are usually implemented and what optimisations are usually included in the HLS flow. Next, we dived into scheduling in HLS and explain how existing approaches exploit hardware parallelism. Additionally, we compared a few static analysis techniques and discuss how they can be used in the HLS optimisations. Finally, we introduced the benchmark sets that are used for evaluating the works in the rest of the thesis.

Chapter 3

Scheduling Overview and DASS Tool

This chapter first extends the publication at TRETS 2022, which formulates the specifications of dynamic scheduling for HLS, and formulates both static and dynamic scheduling for HLS with their implementation in state-of-the-art HLS tools. Second, we present the implementation of DASS HLS compiler expanded from the publications at TCAD 2021 and FPGA 2020.

3.1 Scheduling Formalisation

Here we will first explain the scheduling specifications for HLS. We then illustrate how the specifications are adapted to the control flow graph. Next, we show how the specifications are restricted and implemented in existing HLS tools both for static scheduling and dynamic scheduling. Finally, we show these restricted specifications can be relaxed for potential performance improvement.

3.1.1 Scheduling Specifications

The fundamental specifications of scheduling are mainly the dependence constraints. For two run-time events x and y, we introduce the following terms:

- $d(x, y)$ holds when the executions of x and y have dependences,
- $x \prec y$ holds when x executes before y in strict program order,
- $t(x) \in \mathbb{N}$ denotes the start time of the execution x in clock cycles, and
- $l(x) \in \mathbb{N}$ denotes the latency of the execution x in clock cycles.

The general dependence constraint is listed as follows:

$$\forall x, y.\, d(x, y) \wedge x \prec y \Rightarrow t(x) + l(x) \leq t(y) \tag{3.1}$$

where x and y are the operations in the input program.

The total latency of the execution is defined as t_T:

$$\forall x.\, t_T \geq t(x) + l(x) \tag{3.2}$$

The goal is to determine a schedule with a minimum t_T that does not break Constraint 3.1.

3.1.2 Control Flow Graph

We assume that the input program is sequential. The input program is initially lowered from C/C++ to a control flow graph (CFG), where each basic block (BB) contains instructions in strict program order. For a given program and its inputs, we define the following terms for the input source:

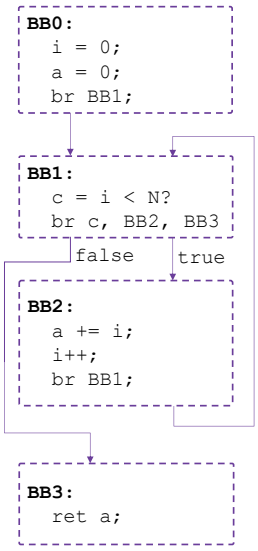
- $B = \{b_1, b_2, ...\}$ denotes the set of all the BBs in the program, and
- $I_b = \{i_1, i_2, ...\}$ denotes the set of all the instructions within a BB b.

The original execution order of the input sequential program can be defined as follows:

- $E_B \subseteq B \times \mathbb{N}$ denotes the set of executions of BBs, where (b_h, n) denotes execution of b_h in its nth iteration,

```
1 int func() {
2   int i, a = 0;
3   for (i=0; i<3; i++)
4     a += i;
5   return a;
6 }
```

(a) Example code.



(b) CFG.

```
1  BB0: // b0(0)
2    a = 0;        // i0(0)
3    i = 0;        // i1(0)
4    goto BB1;     // i2(0)
5  BB1: // b1(0)
6    c = (i < 3)   // i3(0)
7    goto BB2      // i4(0)
8  BB2: // b2(0)
9    a += i;       // i5(0)
10   i++;          // i6(0)
11   goto BB1;     // i7(0)
12 BB1: // b1(1)
13   c = (i < 3)   // i3(1)
14   goto BB2      // i4(1)
15 BB2: // b2(1)
16   a += i;       // i5(1)
17   i++;          // i6(1)
18   goto BB1;     // i7(1)
19 BB1: // b1(2)
20   c = (i < 3)   // i3(2)
21   goto BB2      // i4(2)
22 BB2: // b2(2)
23   a += i;       // i5(2)
24   i++;          // i6(2)
25   goto BB1;     // i7(2)
26 BB1: // b1(3)
27   c = (i < 3)   // i3(3)
28   goto BB3      // i4(3)
29 BB3: // b3(0)
30   return a      // i8(0)
```

(c) Execution order of BBs and instructions.

Figure 3.1: An example of a program and its execution order of BBs and instructions.

- $\prec \subseteq E_B \times E_B$ denotes the original program order of BB execution, where $(b_h, n) \prec (b_{h'}, n')$ denotes (b_h, n) executes before $(b_{h'}, n')$ in strict program order,
- $E_I \subseteq \bigcup_{b \in B} I_b \times \mathbb{N}$ denotes the set of execution of instructions, and
- $\prec_b : I_b \times I_b$ denotes the original program order of instruction execution within a BB b.

By combining $\prec$ and $\prec_b$ lexicographically, the original program order of the instruction execution can be obtained. This is used as a reference for correctness check. A schedule being correct is defined as the execution result by this schedule is always the same as the result of sequential execution in the original program order. The dependence constraint for a CFG is as follows.

$$\forall i, i', n, n'. (i, n) \prec (i', n') \wedge d((i, n), (i', n')) \Rightarrow t(i, n) + l(i, n) \leq t(i', n') \tag{3.3}$$

Figure 3.1a shows a code example, and its CFG is shown in Figure 3.1b. In the figures, a

`for` loop is lowered into four BBs, where $|B| = 4$. Each BB contains a few instructions. The execution order of these instructions is shown in Figure 3.1c. Because an instruction may have multiple iterations, each iteration of an instruction is considered an execution event. For this example, $|\bigcup_{b \in B} I_b| = 9$ and $|E_I| = 21$.

3.1.3 Static Scheduling Implementation

Traditional HLS tools use modulo scheduling for static pipelining of loops or non-control flow functions [105]. In static scheduling, the time constraints and dependences are restricted to simplify the static analysis. Here we only illustrate the formulation of loop pipelining since a non-control flow function could be considered a loop with a single iteration.

In HLS, loop pipelining [47, 106] allows multiple iterations of a loop to be executed concurrently. The same operation in two consecutive iterations has a time difference in clock cycles, also known as the initiation interval (II). A small II leads to high throughput of the hardware design since it allows loop iterations to be executed at an early time. Traditional modulo schedulers are not flow-sensitive and therefore need to make conservative approximations that cover all possible control paths. In order to preserve correctness, the scheduler has to ensure that any data processed by an operation is always valid before the operation starts. Since the exact hardware behaviour can vary, the static scheduler has to assume the "worst case" for loop pipelining, leading to a static value of II.

There are two properties of a loop that limit how small an II can be achieved.

(1) Iteration latency: the latency of each iteration. High iteration latencies can lead to high IIs, especially if an early operation in one iteration has to wait for a late operation in a previous iteration to complete. Iteration latencies can vary at run-time between iterations, but existing static schedulers just take the maximum value.

(2) Dependence distance: the number of iterations that separate an operation from its dependants. Low dependence distances can lead to high IIs, because they limit the number of

iterations that can be safely overlapped. Dependence distances can vary between iterations, but existing static schedulers just take a constant distance of 1 if it is not a constant.

The iteration latency is approximated to a constant by restricting the start times and latencies of executions:

$$t(i,n) = \alpha(i) + Pn, \alpha_i \geq 0 \tag{3.4}$$

$$L(i) = \max_i l(i,n) \tag{3.5}$$

$\alpha(i)$ is a constant for an instruction i as its offset time, P is the static initiation interval, and L is the approximated iteration latency. The model assumes that an instruction i always takes $L(i)$ cycles to execute, and its latency $L(i)$ is approximated to the upper bound of $l(i,k)$. The total execution time of the loop is bounded by

$$t_T = P(N-1) + \max_n(\alpha(i) + L(i)) \approx PN \tag{3.6}$$

where N is the total number of iterations.

The dependence distance is approximated to a constant by restricting the dependence between iterations:

$$\forall i, i', n, n'.\, (i,n) \prec (i',n') \wedge d((i,n),(i',n')) \wedge n \neq n' \Rightarrow d_{\text{dep}} \leq n' - n \tag{3.7}$$

d_{dep} is the minimum dependence distance in the loop. The scheduler conservatively assumes that each iteration has a dependence distance of d_{dep}. The dependence constraint then becomes the following.

$$\forall i, i', n. \wedge d((i,n),(i',n+d_{\text{dep}})) \Rightarrow t(i,n) + l(i,n) \leq t(i',d_{\text{dep}}) \tag{3.8}$$

Substituting the approximation in Constraint 3.4 into Constraint 3.8, the dependence constraint

becomes:

$$\alpha(i') + P(n + d_{\text{dep}}) \geq \alpha(i) + Pn + l(i, n) \tag{3.9}$$

This is then reformulated to:

$$\alpha(i') \geq \alpha(i) + Pd_{\text{dep}} + l(i, n) \tag{3.10}$$

Substituting the approximation in Constraint 3.5 into Constraint 3.10:

$$\alpha(i') \geq \alpha(i) + Pd_{\text{dep}} + L(i) \tag{3.11}$$

$$\geq \alpha(i) + Pd_{\text{dep}} + l(i, n) \tag{3.12}$$

However, the run-time dependence d is hard to capture at compile time. This affects the approximation of both the dependence distance and dependence constraints. In existing HLS tools like Xilinx HLS tools [14] and LegUp [106], the dependence distance is approximated to its minimum value and remains constant for loop pipelining. There are also works investigating how to get a d_{dep} closer to d. For instance, we propose an approach by exploiting the correlation between dependence distance and iteration latency for each dependence using static analysis [107]. This approach is based on purely static scheduling and is not included in this thesis, but it can achieve better performance for some loops compared to traditional static scheduling.

3.1.4 Dynamic Scheduling Implementation

We now introduce the formulation of dynamic scheduling for HLS. Dynamic scheduling also exploits parallelism from a CFG of the program but uses a different scheduling model. As shown in Section 3.1.2, the execution of BBs $\prec$ could be dynamic, where a BB may have a different number of iterations than another BB. However, inside each BB, the execution of instructions $\prec_b$ is static, where they always have the same number of iterations and execution order inside

the BB, as shown in the following constraints.

$$\forall b, i, k.\, (b, k) \in E_B \wedge i \in I_b \Rightarrow (i, k) \in E_I \tag{3.13}$$

$$\forall i, i', k.\, (i, k) \in E_I \wedge (i', k) \in E_I \wedge i \prec i' \Rightarrow (i, k) \prec (i', k) \tag{3.14}$$

The dependences are resolved at run-time. There are two kinds of dependences: memory dependences (*i.e.* dependence via a memory location) and data dependences (*i.e.* dependence via a program variable). There are also two scopes of dependence: between instructions in the same BB, and between instructions in different BBs. This leads to four cases to consider:

(1) Intra-BB data dependences: these can be respected by placing handshaking connections between the corresponding hardware operations in the circuit.

(2) Intra-BB memory dependences: these can be kept in the original program order using *load-store queues* (LSQs) [60]. An LSQ is a hardware component that schedules memory operations at run time.

(3) Inter-BB data dependences: these can be respected using handshaking connections, as in (1), and additionally by starting BBs in strict program order, so that the inputs of each BB are accepted in program order [108].

(4) Inter-BB memory dependences: these can be respected by starting BBs in strict program order and using an LSQ, as in (2).

First, the intra-BB data dependences are represented as edges in data flow graphs (DFGs) inside BBs and can be directly mapped to *handshake signals* between hardware operations. An operation may have variable latency depending on its inputs. Operations which are not connected by handshake signals between them are independent and can execute in parallel or out-of-order. Although an operation may have a variable latency and execute out-of-order, each data path still propagates in-order data through the edges as formalised by Carloni *et al.* [59]. Such a preserved data order inside each BB preserves the intra-BB data dependences.

Second, the intra-BB memory dependences are dynamically scheduled using LSQs. Dynamatic analyses $\prec_b$ and statically encodes the sequential memory order of each BB into the LSQ. An LSQ allocates the memory operations in a BB into its queue at the start of the corresponding BB execution. It dynamically checks these memory operations following the order of $\prec_b$ and executes a memory operation if it is independent of all its priorly executing operations. The LSQ ensures that the intra-BB memory dependences are resolved by statically encoding $\prec_b$ into the LSQ for the dependence check.

We then need to resolve the inter-BB dependences. Completely resolving inter-BB dependences for concurrent execution at run time is still an open question. Dynamatic [13] enables dynamic scheduling with only one restriction. The restriction requires BB executions must start sequentially in strict program order, even if they can execute in parallel and end out-of-order. Let $t : E_I \cup E_B \to \mathbb{N}$ denote the start times of an instruction execution or a BB execution in clock cycles.

$$\forall i, b, k.\, i \in I_b \wedge (b, k) \in E_B \Rightarrow t(b, k) \leq t(i, k) \tag{3.15}$$

$$D : \forall e, e'.\, e \in E_B \wedge e' \in E_B \wedge e \prec e' \Rightarrow t(e) < t(e') \tag{3.16}$$

This preserves the original program order $\prec$ during run-time hardware execution and provides a reference of correctness for dynamic scheduling when combining with $\prec_b$.

Third, with Constraint 3.16, the inter-BB data dependences are preserved using multiplexers. Dynamatic uses a multiplexer to select data input to a BB at the start of the BB by its preceding BB execution. Since the BBs are restricted to start sequentially, there can only be at most one BB receiving at most one starting signal from its multiple preceding BBs. The input data of a BB is selected by the multiplexers based on the starting signal and accept the correct input data for the computation. The starting signal in the sequential BB execution order ensures the data to each BB is also in strict program order. This ensures in-order data flow between BBs, which preserves inter-BB data dependences.

Finally, the inter-BB memory dependences are resolved by the LSQ by dynamically allocating

memory operations between these BB executions. An LSQ dynamically monitors the start signals of BB executions and allocates groups of memory operations in the same BB order, the same definition as $\prec$. It checks the memory operations in an order that combines $\prec$ and $\prec_b$ lexicographically, which is the same as the original program order. As said earlier, with the logic that resolves the intra-BB memory dependences, the LSQ ensures that the out-of-order memory execution always has the same results as the execution in the allocated order *i.e.* the program order. Such an approach of preserving $\prec$ for allocation in the LSQ resolves the inter-BB memory dependences.

With the implementation above, $\prec_b$ is directly encoded to hardware at compile time, and $\prec$ is recovered at run time. All four kinds of dependences can be resolved by checking dependences between operations in strict program order. Let $d : (E_I \cup E_B)^2 \to \{0, 1\}$ denote whether two executions that have dependences, and let $l : E_I \to \mathbb{N}$ denote the latencies of instruction execution in clock cycles. Dynamatic ensures the following dependence constraint always holds:

$$\forall e, e'.\, e \in E_I \wedge e' \in E_I \wedge e \prec e' \wedge d(e, e') \Rightarrow t(e) + l(e) \leq t(e') \tag{3.17}$$

3.1.5 Optimisations for Dynamic & Static Scheduling

Previous sections demonstrate existing static scheduling and dynamic scheduling implementations by restricting the scheduling specifications for CFG execution. However, these formulations either cost large area from supporting flexibility at run time (dynamic scheduling), or performance overhead from conservative control decisions at compile time (static scheduling). Here we show how to achieve a better scheduling solution by combining the best of both worlds. This section shows a set of techniques presented in this thesis that could help improve the hardware design in either area or performance.

Combining Dynamic and Static Scheduling

Static scheduling is known for efficient resource usage because of the predictability of its schedule. We seek to statically schedule part of the dynamically scheduled hardware as internal components, named static islands, for better area efficiency. Such dynamic and static scheduling approach combines the formulation in both Section 3.1.3 and Section 3.1.4.

Let $S = \{s_1, s_2, ...\}$ be the set of all the static islands. We use $s \subseteq \bigcup_{b \in B} I_b$ to denote a static island. Every two static islands must contain disjoint sets of instructions:

$$\forall s, s'. \, s \in S \wedge s' \in S \wedge s \neq s' \Rightarrow s \cap s' = \emptyset \tag{3.18}$$

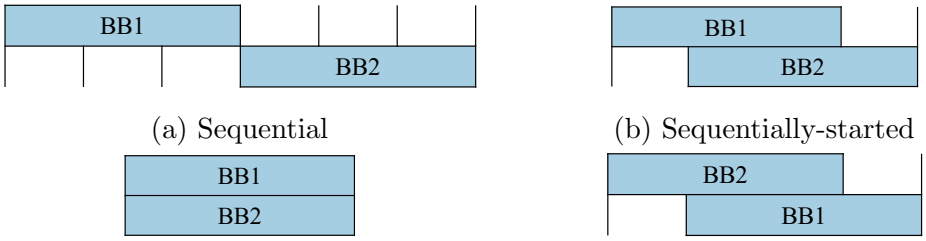
Inside each static island, the constraints for static scheduling must hold. In the dynamically scheduled hardware, the constraints for dynamic scheduling must hold, treating each static island as a single instruction with a longer latency. This approach raises three challenges:

1. How should static islands be correctly and efficiently integrated into their dynamically scheduled surroundings?
2. How should a set of good static islands be automatically determined for better area efficiency and high performance?
3. How should the scheduling constraints of static islands be optimised using the information from their dynamically scheduled surroundings?

In the later section of this chapter, we will illustrate how to overcome challenge 1 and introduce our end-to-end HLS flow named DASS. In Chapter 4, we illustrate how to overcome challenges 2 and 3 using static analysis.

Parallelising Dynamic Control Flow

Constraint 3.16 restricts BBs to start sequentially and reduces the need for static analysis in dynamic scheduling. However, this could be too conservative for dependence analysis. For



(a) Sequential (b) Sequentially-started

(c) Simultaneous (d) Out-of-order

Figure 3.2: Examples of basic block schedules. Assume BB1 executes before BB2 in the original program order. Existing dynamic scheduling only supports (a) and (b). Chapter 5 shows how to achieve (c) and (d) by relaxing constraint 3.16.

instance, when not all the BB executions have dependences, some BBs can start in parallel or out of order, as shown in Figure 3.2. We seek to relax Constraint 3.16 and improve the performance by early starting independent BB executions.

We now demonstrate how Constraint 3.16 can be relaxed in dynamic scheduling with the help of static analysis. Only the BB executions that have dependences must start sequentially. The existence of dependence between two particular BB executions can be defined as whether exists an instruction execution in a BB execution depends on another instruction execution in the other BB execution.

$$d((b,k),(b',k')) = (\exists i, i'.\,(i,k) \in E_I \wedge (i',k') \in E_I \Rightarrow d((i,k),(i',k'))) \tag{3.19}$$

Then Constraint 3.16 can be relaxed to:

$$D' : \forall e, e'.e \in E_B \wedge e' \in E_B \wedge e \prec e' \wedge d(e,e') \Rightarrow t(e) < t(e') \tag{3.20}$$

However, analysing the run-time dependence between two BB execution at compile time is still challenging. In order to enable static analysis for run-time dependence analysis, we over-approximate the case for two particular BB executions to the case for all the executions of two particular BBs. Let $d' : B \times B \rightarrow \{0,1\}$ denote whether two basic blocks may have dependence

during their executions.

$$d'(b, b') = (\exists k, k'.\, d((b, k), (b', k'))) \tag{3.21}$$

We then relax Constraint 3.20 to the following:

$$D'' : \forall b, b', k, k'.\, (b, k) \in E_B \wedge (b', k') \in E_B \wedge (b, k) \prec (b', k') \wedge d'(b, b') \Rightarrow t(b, k) < t(b', k') \tag{3.22}$$

This means BBs that cannot have dependence during the whole execution can start in parallel or out of order. The dependence set of the program from our formulation lies between Constraint 3.16 and Constraint 3.20, where $D' \subseteq D'' \subseteq D$. With this new constraint, the dependences are still respected with existing multiplexers and LSQs since the dependent BB executions remain to start sequentially.

In Chapter 5, we demonstrate how to apply static analysis for relaxing Constraint 3.16 towards Constraint 3.22 and enable two hardware optimisation techniques, C-slow pipelining and inter-BB parallelism, for dynamic-scheduling HLS.

3.2 DASS HLS Tool

In this section, we demonstrate how to use formalisation in Section 3.1 to implement a HLS tool named Dynamic & Static Scheduling (DASS) that supports statically scheduled components in dynamically scheduled hardware as static islands: a marriage of static scheduling and dynamic scheduling that aims for minimal area *and* maximal performance, as sketched in Figure 1.1.

In Section 3.2.1, we compare static scheduling and dynamic scheduling in detail, and introduce the motivation of DASS. In Section 3.2.2, we give a working example to motivate the combined scheduling approach in which some scheduling decisions are taken dynamically at run-time and the others are determined offline using traditional HLS techniques. In Section 3.2.3, we describe how our proposal overcomes challenges related to component integration and shared memory.

Section 3.2.3 details a prototype implementation of DASS that uses Xilinx Vivado HLS [14] for static scheduling and Dynamatic [13] for dynamic scheduling. In Section 3.2.4, we evaluate the effectiveness of DASS on a set of benchmarks and compare the results with the corresponding statically scheduled-only circuits and dynamically scheduled-only circuits.

3.2.1 Introduction

Here we first discuss the benefits of static scheduling and dynamic scheduling in detail. The advantage of static scheduling is that since the hardware is not yet online, the scheduler has an abundance of time available to make good decisions. It can seek operations that can be performed simultaneously, thereby reducing the latency of the computation. It can also adjust the start times of operations so that resources can be shared between them, thereby reducing the area of the final hardware. However, a static scheduler must make conservative decisions about which control-flow paths will be taken, or how long variable-latency operations will take, because this information is not available until run-time.

Dynamic scheduling, on the other hand, can take advantage of this run-time information. Dynamically scheduled hardware consists of various components that communicate with each other using handshaking signals. This means that operations are carried out as soon as the inputs are valid. In the presence of variable-latency operations, a dynamically scheduled circuit can achieve better performance than a statically scheduled one in terms of clock cycles. However, these handshaking signals may also cause a longer critical path, resulting in a lower operating frequency. In addition, because scheduling decisions are not made until run-time, it is difficult to enable resource sharing. Because of this, and also because of the overhead of the handshaking circuitry, a dynamically scheduled circuit usually consumes more area than a statically scheduled one.

The basic idea of DASS is to identify the parts of an input program that may benefit from static scheduling—typically parts that have simple control flow and fixed latency—and to use static scheduling on those parts. In this section, it is the user's responsibility to annotate these parts of the program using pragmas, and the automation of this process is explained in Chapter 4.

```
1  // initialised at run-time to {1, -1, 1, -1, ...}
2  double A[N];
3
4  double g(double d) {
5    return (((((d+0.2)*d+0.3)*d+0.6)*d+0.2)*d+0.7)*d+0.2;
6  }
7
8  double filterSum() {
9    double s = 0.0;
10   for (int i = 0; i < N; i++) {
11     double d = A[i];
12     if (d > 0) {
13       double t = g(d);
14       s += t;
15     }
16   }
17   return s;
18 }
```

Figure 3.3: Motivating example for DASS.

The static islands are then treated as black boxes when applying dynamic scheduling on the remainder of the program.

Several challenges must be overcome to make this marriage work. These include:

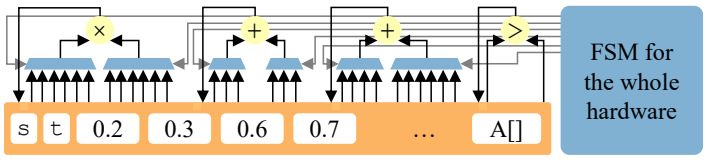
1. How should statically scheduled parts be correctly and efficiently integrated into their dynamically scheduled surroundings?

2. How should the memory be correctly and efficiently shared between a static island and the dynamically scheduled circuit?

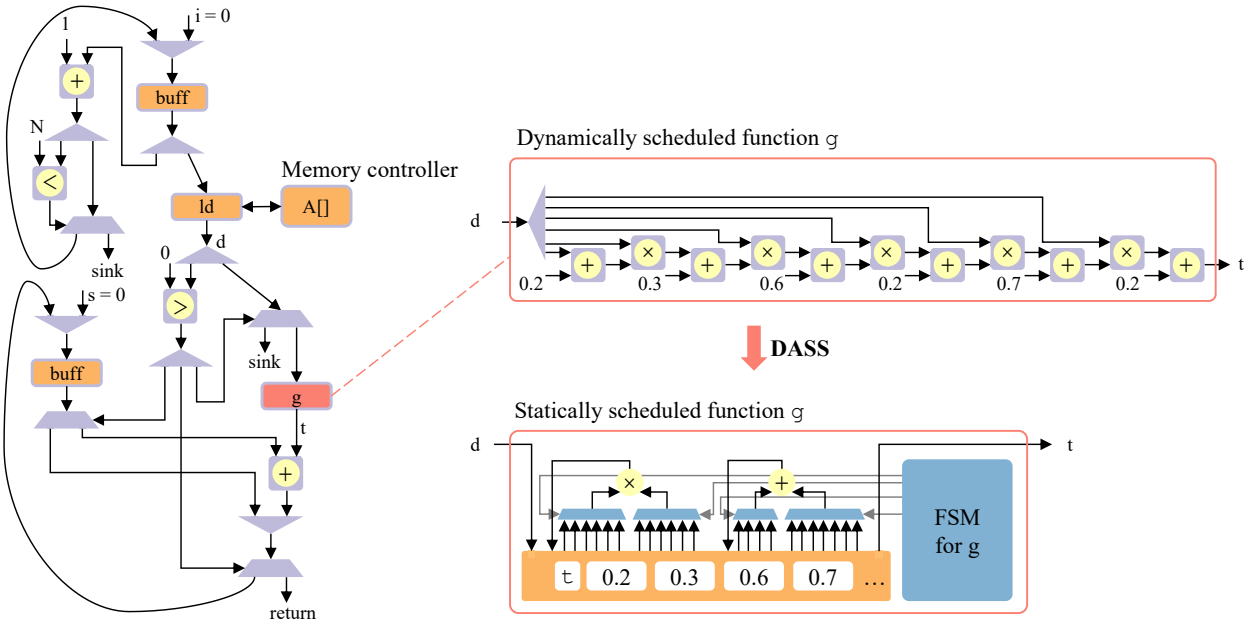
In the following sections, we show how these challenges can be overcome. Our evaluation on several realistic benchmarks demonstrates that it is possible to obtain up to 74% of the area savings that would be made by switching from dynamic scheduling to static scheduling and up to 135% of the performance benefits that would be made by switching from static scheduling to dynamic scheduling. In other words, DASS can obtain most of the area benefits associated with static scheduling, and can actually outperform both dynamic scheduling and static scheduling.

3.2.2 Motivating Example

We now demonstrate our approach via a worked example.



(a) Statically scheduled circuit.



(b) Dynamically scheduled circuit and its transformation into a DASS circuit.

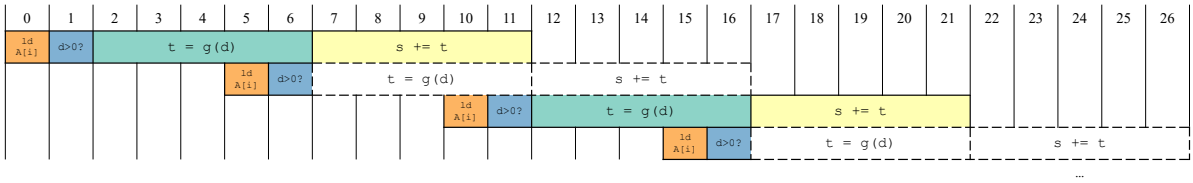
Figure 3.4: Hardware generated from dynamic and static schedules for the example in Figure 3.3. Dynamic scheduling has a larger area when compared with static scheduling, and DASS makes the area smaller without losing performance.

Figure 3.3 shows a simple loop that operates on an array `A` of doubles. It calculates the value of `g(d)` for each non-negative `d` in the array, and returns the sum of these values. The `g` function represents the kind of high-order polynomial that arises when approximating complex non-linear functions such as tanh or log. If the values in `A` are provided at run-time as shown at the top of Figure 3.3, then function `g` is only called on odd-numbered iterations.

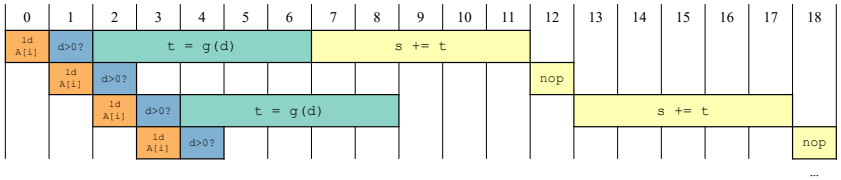
To synthesise this program into hardware, we consider three scheduling techniques: static scheduling, dynamic scheduling, and our approach, DASS.

Static scheduling – Small Area but Low Performance

The hardware resulting from static scheduling is shown in Figure 3.4a. It consists of three main parts. The orange block at the bottom is the registers and memory blocks that store the



(a) The schedule by static scheduling (the loop II = 5).



(b) The schedule by dynamic scheduling or DASS (II_g in DASS = 1).

Figure 3.5: Dynamic and static schedules for the example in Figure 3.3. The latency of function `g` is 59 cycles but is represented as 5 cycles in the figure to save space. Dynamic scheduling has better performance when compared with static scheduling. DASS has a schedule with comparable performance to dynamic scheduling because it has the same schedule.

data. At the top are several operators that perform the computation described in the code. On the right is a finite-state machine (FSM) that monitors and controls these data operations in accordance with the schedule determined by the static scheduler at compile time. The static island achieves good area efficiency through the use of resource sharing, *i.e.* using multiplexers to share a single operator among different sets of inputs.

The timing diagram of the static island is shown in Figure 3.5a. It is a pipelined schedule with an initiation interval (II) of 5. The II cannot be reduced further because of the loop-carried dependency on `s` in line 14. Since the `if` decision is only made at run-time, the scheduler cannot determine whether function `g` and the addition are performed in a particular iteration. It therefore conservatively reserves their time slots in each iteration, keeping II constant at 5. This results in empty slots in the second and fourth iteration (shown with dashed outlines in the figure), which cause the operations in the next iteration to be unnecessarily delayed.

Dynamic Scheduling – Large Area but High Performance

The dynamically scheduled hardware is a dataflow circuit with a distributed control system containing several small components representing instruction-level operations [13], as shown in Figure 3.4b. Each component is connected to its predecessors and successors using a hand-

Table 3.1: Elastic components for dynamically scheduled HLS.



Merge: takes the input data from an arbitrary predecessor and propagates it to its single successor.



Fork: takes the input data from its single predecessor and replicates it to each of its multiple successors.



Join: triggers its single successor only when the input data of its all predecessors is available.



Branch: takes the data from its data predecessor and propagates it to one of its multiple successors based on the select value from its control predecessor.

shaking interface. This handshaking, together with the inability to perform resource sharing on operators, causes the area of the dynamically scheduled hardware to be larger than the corresponding statically scheduled hardware. The components used in this example are listed in Table 3.1.

The timing diagram of the dynamically scheduled hardware is shown in Figure 3.5b. It has the property that each operator executes as soon as its inputs are valid so that the throughput can be higher than that for statically scheduled hardware. For instance, it can be seen that the read of `A[i]` in the second iteration starts immediately after the read in the first iteration completes. Most stalls in dynamically scheduled hardware are due to data dependences. For instance, the execution of function `g` and the addition in the second iteration are skipped as `d = -0.1` $<$ `0`, leading to `s = old_s`. The operation is not carried out immediately after the condition check but stalled until `s += t` in the first iteration completes, since it requires the output from the previous operation as input. Then it is immediately followed by `s += t` in the third iteration.

DASS – Both Small Area and High Performance

The DASS hardware combines the previous two scheduling techniques. It is based on the observation that although the overall hardware performance benefits from dynamic scheduling, the function `g` does not because it has a fixed latency. Therefore, we replace the dataflow implementation of `g` with a functionally equivalent statically scheduled implementation, *i.e.* a static island. The static island uses resource sharing to reduce six adders and five multipliers

down to just one of each. The rest of the hardware outside `g` is the same as the dynamically scheduled hardware. Because `g` represents a substantial fraction of the overall hardware, this transformation leads to the area of the DASS hardware being close to that of the pure statically scheduled hardware, as shown in Figure 1.1.

The timing diagram of the DASS hardware is the same as that of the dynamically scheduled hardware, as shown in Figure 3.5b.[1] In the dynamically scheduled hardware, `g`'s schedule is determined at run-time, while in the DASS hardware it is determined by the static scheduler; in both cases, the timing diagram is the same. The data-dependent `if` condition in the loop remains part of the dynamically scheduled hardware to maximise throughput. Hence the DASS hardware and the dynamically scheduled hardware have the same throughput in terms of clock cycles. However, since static scheduling optimises the critical path in `g` using hardware retiming, the DASS hardware can actually run at a higher clock frequency. Therefore, in this example, DASS hardware achieves not merely the 'best of both worlds', but actually achieves *better* performance than dynamically scheduled hardware (in terms of wall clock time), and *comparable* area to statically scheduled hardware, as shown in Figure 1.1.

In the following sections, we give the details of how to configure the constraints of the static islands for maximising resource sharing and preserving performance, and the methodology for integrating the static islands into the dynamically scheduled hardware.

3.2.3 Methodology

In this section, we show how to partition and synthesise some functions into static islands and the rest of the program into dynamically scheduled hardware. We first discuss which programs are amenable to our approach. We then detail the integration of the static islands into the dynamically scheduled hardware using a dedicated wrapper, which ensures that the data is correctly propagated between these two architectures. Finally, we show how to enable memory sharing among static islands and dynamically scheduled hardware correctly.

[1] Actually, the latency of function `g` varies slightly between dynamic scheduling and static scheduling for technical reasons, as explained in Section 3.2.4.

Applicability of Our Approach

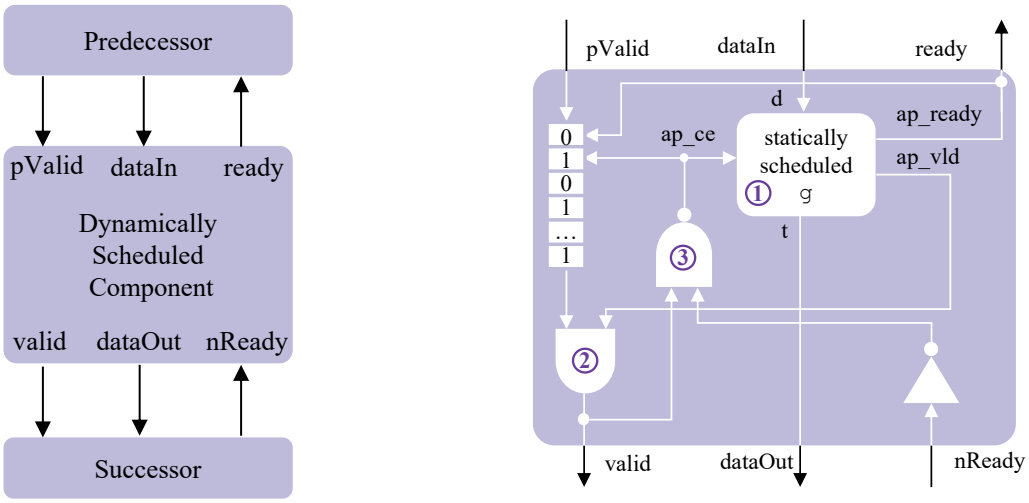
Our approach is generally applicable, in the sense that it can be used wherever static scheduling or dynamic scheduling can be used. The following conditions indicate scenarios where our approach is likely to yield the most substantial benefits over static scheduling and dynamic scheduling:

1. there is an opportunity to improve throughput using information that only appears at run-time,
2. at least one region of the code has a constant (or low variability) latency, and
3. this code region has an opportunity for resource sharing.

The first condition indicates that the design may be amenable to dynamic scheduling. The second and third reflect the fact that static scheduling determines a fixed schedule and can take advantage of resource sharing. We emphasise that not *all* of the conditions above need to hold for an input program to benefit from our approach; it is simply that each condition listed above is desirable.

Integrating Static Islands

We now show how to implement a wrapper for a static island around the dynamically scheduled surroundings. A wrapper design of a synchronous circuit around latency-insensitive designs is explained by Carloni [59]. In the design, the wrapper always fires as long as inputs are valid and the successor is ready. In other words, they assume no stall is caused by the internal architecture of the synchronous circuit, *i.e.* the II of the synchronous circuit is always 1. However, in most hardware designs, we want to have efficient hardware architecture as well as high performance. II greater than 1 allows designers to perform more hardware optimisation, like resource sharing, while still preserving high performance. In our work, we support and prefer that II greater than 1, so resource sharing is possible.



(a) Handshaking interface in a dynamically scheduled circuit.

(b) Statically scheduled function g as a dynamically scheduled component.

Figure 3.6: The static island g is wrapped with additional control circuitry for interfacing to the dynamically scheduled circuit.

A dynamically scheduled circuit is constructed as a dataflow circuit, containing a number of small components, while a static island has a centralised FSM for control. We regard each static island as a component in the dataflow circuit, as indicated in Figure 3.4a. In this section, we explain how to make a static island behave like a dynamically scheduled component so that it can be integrated into the overall dynamically scheduled hardware. Let us look at function g in Figure 3.3, for example, which is a single-input and single-output function. The multiple-input and multiple-output cases will be discussed shortly.

In the dynamically scheduled part, each component communicates with its predecessors and successors using a set of handshaking signals as shown in Figure 3.6a. Each dynamically scheduled component uses the bundled data protocol [109] for communication, where each data connection has request and acknowledgement signals. For instance, the following is the control interface of a component from Dynamatic [13]:

- *pValid:* an input signal indicating that the data from the predecessor is valid,
- *valid:* an output signal informing the successor that the data from the current component is valid,
- *nReady:* an input signal indicating that the successor is ready to take a new input, and

- *ready:* an output signal informing the predecessor that the current component is ready to take a new input.

On the other hand, traditional statically scheduled hardware has a different interface to monitor and control the states of the centralised FSM. An example of an HLS tool that generates statically scheduled hardware is Vivado HLS [14]. For a typical control interface of a function synthesised into statically scheduled hardware, its control interface is as follows:

- *ap_ce:* The clock enable signal controls all the sequential operations driven by the clock.
- *ap_ready:* The ready signal from the statically scheduled hardware indicates that it is ready for new inputs.
- *ap_vld:* The valid signal indicates the output from statically scheduled hardware at the current clock cycle is valid.

The interface of a static island is not compatible with the above handshaking signals in dynamically scheduled hardware. To overcome this issue, we add a wrapper around each static island, ensuring that the data propagates correctly between the static island and its dynamically scheduled surroundings. This wrapper is generated in two steps:

1. In a static island, any output is only valid for one clock cycle. We design a *valid* signal to correctly send the data to the successor and preserve the output when backpressure from the successor occurs;
2. Since there may be a pipeline stall caused by this component, we design a *ready* signal to send the backpressure to the predecessor, ensuring any valid input is not lost.

We now discuss those two steps in more detail.

Constructing the *valid* signal

In statically scheduled hardware, where the entire schedule is determined at compile-time, the arrival time of each input can be predicted. However, this is not the case in dynamic scheduling

as the behaviour of the rest of the dynamically scheduled hardware is unknown. Two choices are available:

1. stalling the static island until valid input data is available, or
2. letting the static island continue to process data actively in its pipeline, marking and ignoring any invalid outputs.

Since the static island does not have the knowledge of the rest of the dynamically scheduled hardware, the first approach may cause unnecessary stalls. Hence, for performance reasons, we take the second choice with power overhead from keeping static islands active.

An invalid input read and processed by the static island is named a *bubble*. We use a shift register to tag the validity of the data and propagate only the valid data to the successor, as shown in Figure 3.6b. The shifting operation of the shift register is controlled by the state of the static island to synchronise the data operations in the static island. It shifts to the right by one bit every time the static island takes a new input, as indicated by the *ap_ready* signal. The new bit represents whether the newly taken input data is valid or not. A zero represents a *bubble* and a one represents a valid input. The length of the shift register is determined by the latency and the II of function g: $\lceil latency/II \rceil$, where these time constraints are obtained from the scheduling report by the static scheduler. This ensures that when the output is available from the static island, as indicated by the *ap_vld* signal, its validity is indicated by the oldest bit of the shift register. By checking the oldest bit value, only the valid data is propagated to the successor with the *valid* signal high. In summary, we use the shift register to monitor and control the state of the static island, such that the data can be synchronised between the island and dynamically scheduled hardware, filtering out the bubbles to ensure the correctness of the function. Similarly, only the memory operations with valid data are carried out.

Constructing the *ready* signal

The *valid* signal for the successor and the shift register allows data to propagate from the predecessor, through the static island, to the successor. However, the component is not able

to deal with any backpressure from the function or its successor. Backpressure happens when a component is unable to read an input even though it is valid, resulting in its predecessor stalling. In dynamically scheduled hardware, this issue is solved using handshake signals—the hardware stalls when its output is valid but the successor is not ready, as indicated by its *nReady* signal or the *ready* signal from the successor. We design a control circuit to handle the backpressure between dynamically scheduled hardware and a static island. Backpressure can arise between dynamically scheduled hardware and a static island in two ways, which we now discuss.

Backpressure from a static island to its predecessor: In this case, the *ready* signal indicates whether the static island is ready to take an input so *ap_ready* is directly connected to the *ready* signal of the wrapper. It sends feedback to the predecessor such that the predecessor can be stalled, holding the valid input to the static island until the static island is ready.

Backpressure to a static island from its successor: Since the static island holds the output for one cycle when running, we stall the process in the static island to preserve the output data. This is achieved by disabling the clock signal, $ap_ce = 0$, so the static island stops all the sequential processes, preserving the output. The condition for such a stall to occur is that the next output from the static island is valid ($valid = 1$) but the successor is not ready ($nReady = 0$). The static island continues running after the *nReady* signal is set to high, indicating that the successor is ready to accept the output data of the static island. This additional circuitry ensures that the data exchanged between a static island and dynamically scheduled hardware is not lost when any stall occurs.

Handling multiple inputs and multiple outputs

The example above shows the wrapper for a function with a single input and an output. However, it is also common to have a function with zero or more than one input or output. If there is no input and output, the external dynamically scheduled hardware would have no corresponding data port for the component, hence no corresponding handshaking signal is needed. Here we focus on the cases of multiple inputs and outputs.

For multiple inputs, we construct a set of handshaking signals as shown in Figure 3.6b for each input and synchronise data with the help of *join* components in Table 3.1, such that the static island always takes all the input simultaneously. A *join* component is used to preserve the valid inputs to the static island until all the inputs are valid. This is similar to a dynamically scheduled component with multiple inputs. Using *join* components is a simple solution to synchronise the inputs. An optimised approach that uses input offsets will be proposed in section 4.1.

For multiple outputs of the static island, each component has its own handshaking signals. The output handshaking signals are implemented in two parts, the *valid* and *nReady* signals. First, each output has its own *valid* signal. For each output, the static island has a *ap_vld* signal indicating whether the corresponding output is valid. Each *ap_vld* signal is *AND*ed with the oldest bit of the shift register ($sr(0)$) as the corresponding *valid* signal: $valid_i = ap_vld_i \wedge sr(0)$, for $i = 0, 1, 2, ...$ Second, any unset *nReady* signal from the output when the data is valid can disable the clock in the static island as back pressure: $ce = \neg(\vee_i((\neg nReady_i) \wedge valid_i))$.

Shared Memory Between Static Islands and Dynamically Scheduled Hardware

Statically scheduled hardware has its memory accesses pre-scheduled at compile time and can interact with the memory at run time in a predictable sequence; the dynamically scheduled hardware requires a load-store queue (LSQ) [60] that schedules the memory accesses at run-time before accessing the data. In DASS hardware, a combination of the two memory architectures needs to be handled. An LSQ is beneficial for programs that have irregular memory accesses. It does not bring performance improvements when implementing regular computation in the form of a static island. The Dynamatic tool uses static analysis, such as polyhedral analysis, to identify the memory accesses in the dataflow graph that cannot cause any memory conflicts [61]. Then these memory access nodes are directly connected to the BRAM through an arbiter, as the memory controller, instead of being scheduled by an LSQ.

Figure 3.7 shows an example of the memory architecture of purely dynamically scheduled hardware. The nodes on the leftmost are the memory nodes, and the blocks with names of

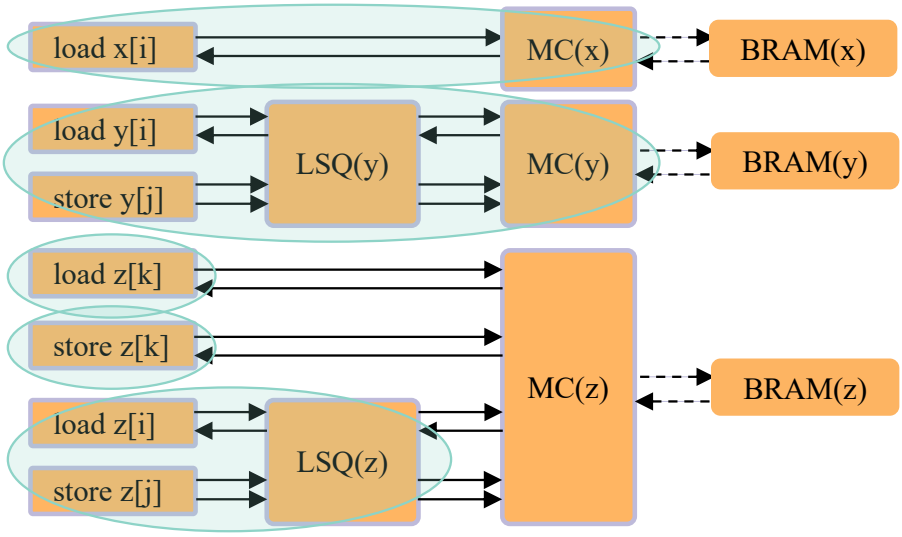


Figure 3.7: An example of the netlist of all memory accesses for dynamically scheduled hardware. The analysis in Dynamatic tool flow already reports which memory node may cause memory conflicts. Each green circle means the minimum unit to be integrated in a static island.

"LSQ" and "MC" are the memory components in the dynamically scheduled hardware. The blocks on the rightmost are the BRAM blocks on FPGAs. Each BRAM block represents an array in the input program. The solid edges represent the handshaking interface, while the dashed edges represent the block memory interface. In dynamically scheduled hardware, each memory node performs a single load/store operation, and all these nodes are connected to the memory components through handshaking signals. The LSQ schedules the memory with dependencies, while the memory controller (MC) is a simple memory arbiter which issues independent memory accesses to memory. Then the memory components serialise the requests from these nodes and perform the corresponding memory operations with the BRAM through the block memory interface.

In the figure, the dynamically scheduled hardware has seven memory access nodes targeting three arrays. Here we assume `load z[k]` cannot have conflicts with other accesses to `z`, the same as `store z[k]`. Since these arrays are separate, there are three memory controllers to manage the accesses to the memory. Firstly, the array `x` is only accessed by a single *load* so the node can be directly connected to the memory controller. Secondly, the *load* and *store* with the indices `i` and `j`, whose values are determined at run time, may depend on each other when accessing the same array `y`. Therefore, an LSQ is required to ensure that those memory accesses are

carried out in the correct sequence. Finally, the memory accesses to array `z` have both regular and irregular patterns. The analysis in Dynamatic proves that the regular memory accesses do not conflict with the irregular memory accesses. Hence, these regular memory accesses can be safely connected to the memory controller, skipping the LSQ. One advantage of such an approach is to minimise the overhead caused by the LSQ and maintain the dynamic mechanism of the memory architecture. The dynamically scheduled compiler analyses the program and constructs an efficient memory architecture above for the dynamically scheduled hardware.

In DASS, we use the results of the above analysis to identify whether the memory accessed by the static islands can share safely with the dynamically scheduled hardware. In the source, we inline all the statically scheduled functions at the top-level program and send it to the memory analyser used by Dynamatic. Dynamatic outputs a memory architecture graph at the top level, such as Figure 3.7. The graph shows the connection of each memory node to the memory controller if the whole hardware is dynamically scheduled. Based on that graph, our tool divides all these memory nodes into several node sets, shown as green circles. All the nodes of a single set must belong to the same hardware region. For a static island that contains unpredictable memory accesses, the top-level static island cannot be pipelined. The reason is that the external dynamically scheduled surroundings ignore the memory dependencies inside the static island. For instance, a static island in a loop may require the maximum loop throughput to be 0.2 in Figure 3.5a, while the dynamically scheduled loop feeds data at a throughput of 1 set of data per clock cycle in Figure 3.5b as it does not see the inter-iteration dependency hidden in the static island.

For instance, in Figure 3.7, all the nodes are divided into 5 sets indicated by the green circles: 1) Any node directly connected to the memory controller is a single set; 2) any node sharing the same LSQ belongs to the same set. The memory accesses in each green circle may cause conflicts, so they have to be scheduled together either dynamically or statically. The reason is that each static island is treated as a black box by the LSQ, and the behaviour of a static island accessing the memory is unknown for a single input. This is different from purely dynamically scheduled hardware, in which each memory node only accesses the memory once every time the single node is triggered. A static island may access memory multiple times in one computation if

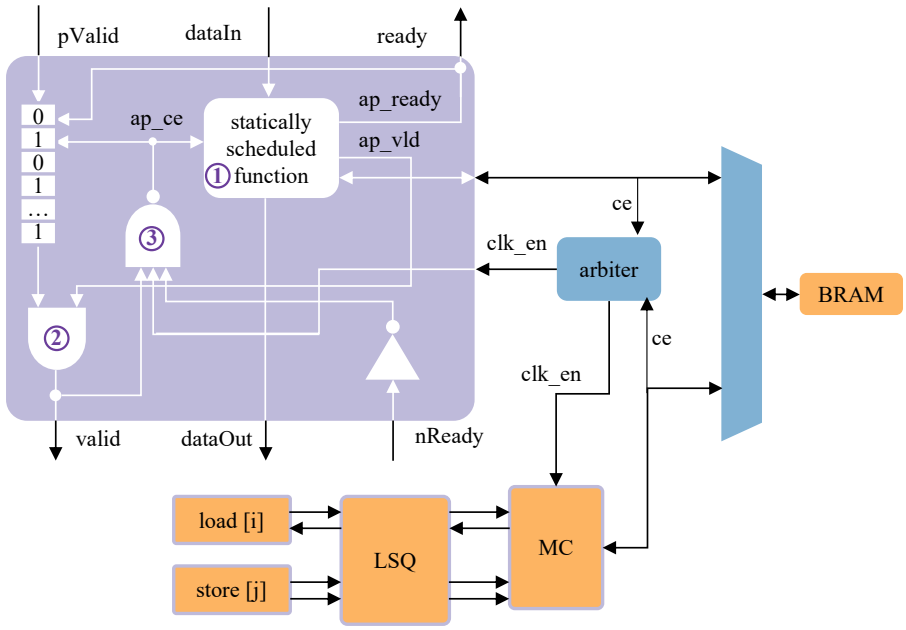


Figure 3.8: Shared memory architecture in the DASS hardware. An arbiter is used to decide which hardware to access the memory in each clock cycle.

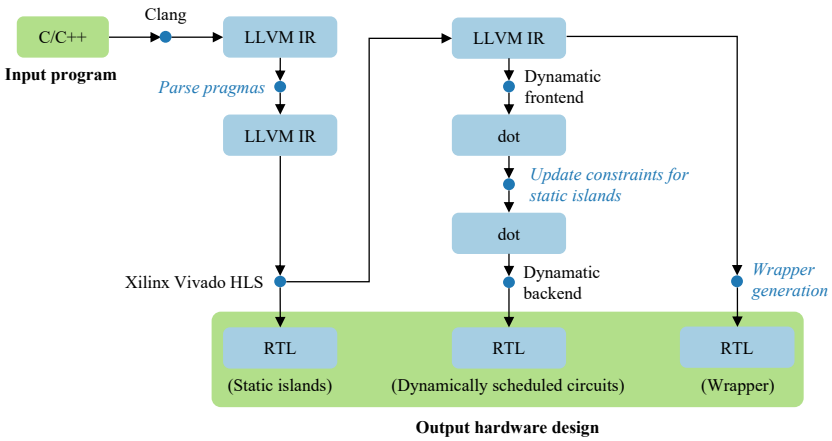
it contains multiple memory statements or loops. Therefore, our approach only allows complete sets in the static island. Such a method ensures that there is no memory conflict between the static island and the dynamically scheduled surroundings whenever they access the shared memory. If a set of nodes using an LSQ is considered to be statically scheduled, the LSQ is no longer needed. The performance may be affected by the conservatism in static scheduling, however, there is a significant area reduction as the LSQ is no longer needed.

Constructing Shared Memory Interface in DASS

In static scheduling, the nodes in Figure 3.7 are no longer distributed but scheduled at compile time. The memory interface is the block memory interface instead of the handshaking signals. Once the access to a shared memory block occurs, our tool automatically adds the memory interface to the wrapper. Figure 3.8 shows an example of a shared memory architecture in the DASS hardware. The yellow block on the left is the wrapper for the static island, as shown in Figure 3.6b. Apart from that, an additional memory wrapper is added, shown as the circuits on the right. As mentioned previously, the static island directly accesses the memory through the BRAM interface shown as the thick arrows. The dynamically scheduled hardware uses

```
1  g(...){
2  #pragma DASS SS II=N
3    ...
4  }
5
6  f(...){
7    ...
8    call g;
9    ...
10 }
```

(a) Input program.



(b) DASS tool flow.

Figure 3.9: With user-specified constraints in pragmas, our tool automatically generates a combined dynamically and statically scheduled circuit.

the memory controllers to transform the handshaking signals in dotted arrows to the same interface as the static island. To ensure that there is no conflict between these two systems, our tool adds a memory arbiter to the memory system. In every clock cycle, the arbiter grants access to either the dynamically scheduled hardware or a static island, and stalls the others by controlling the clock enable bits *ce* of these components. If the race condition happens between a static island and dynamically scheduled hardware or between two static islands, the memory arbiter chooses one to grant in a round-robin fashion and stalls the rest of the hardware for data synchronisation. The priority of the static islands is always higher than the dynamically scheduled hardware since we expect the static islands to run at the highest throughput.

In summary, we identify code that is amenable for DASS, where the design quality of the resulting hardware can be improved. With our wrapper, a static island can work correctly in dynamically scheduled hardware. Finally, we show how to automatically validate the memory correctness between the static islands and dynamically scheduled hardware, and synthesise efficient shared memory architecture in the DASS hardware. Our experiments have shown that the proposed shared memory interface allows us to remove all the LSQs in the benchmarks in Section 3.2.4.

Tool Flow

Our approach is generic and can be used with various static-scheduling and dynamic-scheduling HLS tools. For our work, we choose Vivado HLS [14] and Dynamatic [13] to synthesise static islands and dynamically scheduled hardware respectively. Our tool flow is shown in Figure 3.9. The user-defined scheduling constraints are configured using *pragmas*. DASS takes the input C++ code and splits the functions into two groups based on the pragmas specified by the user, representing the statically scheduled function and the dynamically scheduled functions.[2] We synthesise a function without any scheduling constraints to dynamically scheduled hardware by default. Our tool supports the integration of multiple static islands into a dynamically scheduled function.

A front-end analysis is carried out to identify whether there is inter-iteration dependence or shared memory between the static island and the dynamically scheduled function. Then each static island is synthesised by Vivado HLS. If a static island has no inter-iteration dependence, the II of the function is either an II defined by the user or the optimal II determined by Vivado HLS. If a static island has an inter-iteration dependence, then it is synthesised with a sequential schedule. The resultant static island is then automatically wrapped up to ensure compatibility with the dynamically scheduled hardware interface, as described in Section 3.2.3. Each input variable or output variable of the function is constructed as a data port with a set of handshaking signals. The memory port for exclusive array accesses from the static island is directly forwarded to the memory block. If the array is shared by other hardware, then an arbiter is generated to serialise the memory accesses.

In the dynamically scheduled function that contains static islands, each static island appears as a single component in the dynamically scheduled hardware netlist. We access the dataflow graph in Dynamatic that contains the timing constraints of all these dynamically scheduled components and update the II and latency of each static island in terms of the corresponding scheduling report from Vivado HLS. This ensures correct hardware optimisation in the backend of Dynamatic. Finally, the resultant RTL files represent the final DASS hardware.

[2] A static island is required to be a function, such that it can be scheduled by Vivado HLS.

3.2.4 Experiments

We evaluate our work on DASS using a set of benchmarks explained in Section 2.5, comparing with the corresponding statically scheduled designs and dynamically scheduled designs. We assess the impact of DASS on both the circuit area and the wall clock time.

We evaluate our approach on the total latency which reflects the overall throughput and the area of the whole hardware compared to existing scheduling approaches. Specifically, we select a number of benchmarks where dynamic scheduling generates hardware with lower latency than static scheduling, and show how the area overhead can be reduced while preserving low latency. To ensure fairness, we present the best statically-scheduled solution from Vivado HLS and the best dynamically-scheduled solution from Dynamatic for each benchmark as baselines. In addition, we assume that the designer has no knowledge of the input data distribution for the DASS hardware and show that the area and execution time can still be reduced. This means we use the conservative II automatically obtained from Vivado HLS, *i.e.* the smallest possible II determined by only the topology of the circuit. The timing results of our work are shown as a range of values that depends on the input data distribution. We obtain the total clock cycles from ModelSim 10.6c and the area results from the Post & Synthesis report in Vivado. The FPGA family we used for result measurements is xc7z020clg484 and the version of Vivado software is 2018.3. All our designs are functionally verified in ModelSim on a set of test vectors with input data distributions representative of typical workloads.

Overall Experimental Results

In most benchmarks, our approach has less area and execution time than the corresponding dynamically scheduled hardware. Figure 3.10 shows the overall design quality of our approach compared to the statically scheduled and dynamically scheduled solutions, complementing the detailed results in Table 3.2. In the figure, we show three arrows for each benchmark: the best case (all inputs take the short path), the worst case (all long), and a middle case (half short, half long). The axes are normalised to the corresponding statically scheduled solutions

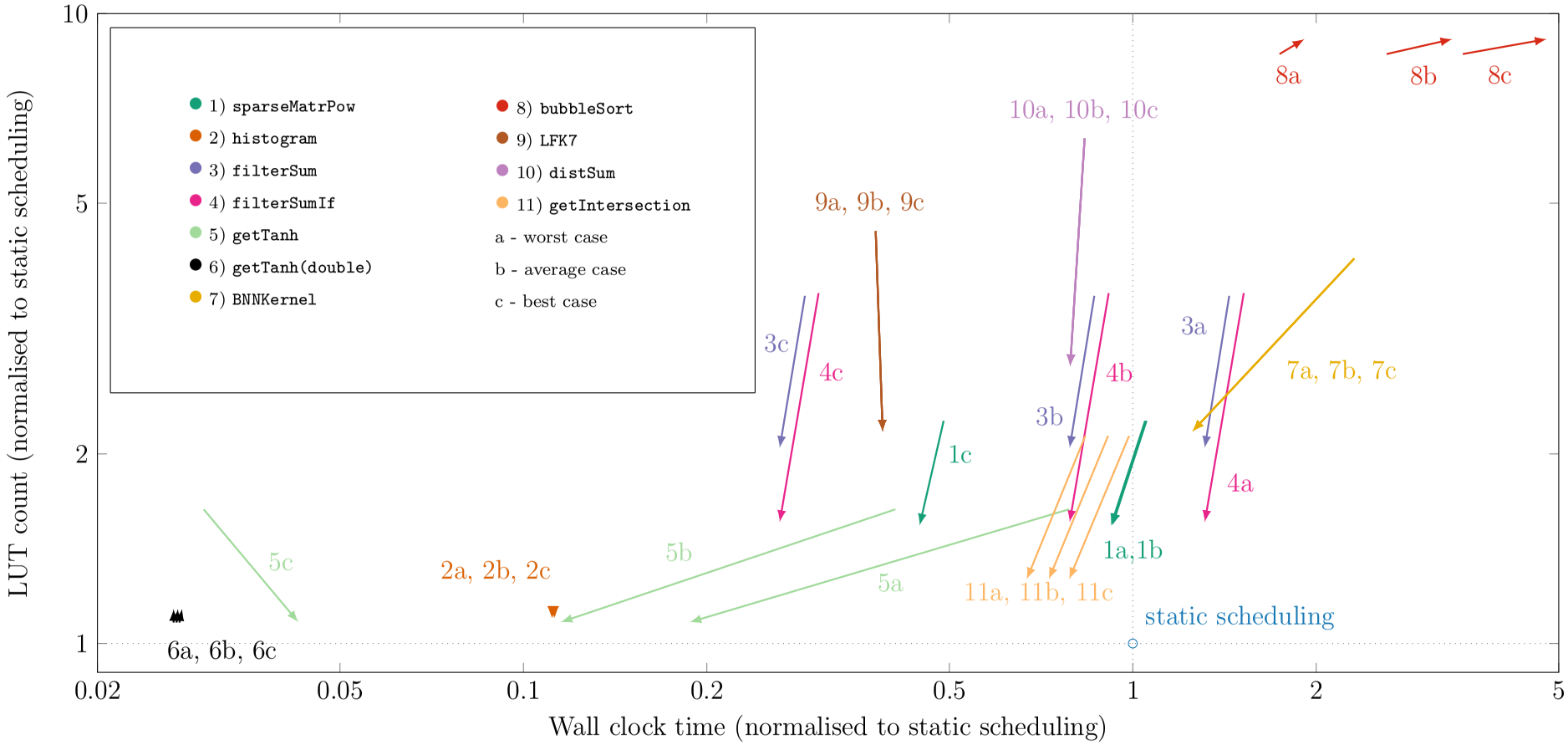


Figure 3.10: The overall effects of our approach over the eleven benchmarks from Table 3.2. Each benchmark is given three different data distributions: (a) worst case, (b) average case, and (c) best case. Each arrow shows a change in area and performance by switching from dynamic scheduling to DASS, both relative to static scheduling. Most of the arrows lie entirely in the second quadrant, which means both dynamic scheduling and DASS are faster but larger than static scheduling. Arrows that point left mean that DASS is faster than dynamic scheduling; arrows that point down mean that DASS is smaller than dynamic scheduling.

Table 3.2: Evaluation of design quality of DASS over eleven benchmarks. Assuming the data distribution is unknown, the II of the static function in DASS is selected as the II in the worst case. The average values are taken except `bubbleSort` as it is not amenable for DASS. DS = dynamic scheduling, and SS = static scheduling.

	II	LUTs			DSPs			Registers			Total Cycles			Fmax/MHz			Wall Clock Time/μs		
		SS	DS	DASS	SS	DS	DASS	SS	DS	DASS	SS	DS	DASS	SS	DS	DASS	SS	DS	DASS
● 1	299	206	465	317	3	6	3	191	489	198	306-30.7k	141-30.0k	141-29.7k	64.9	60.2	67.8	4.7-473	2.3-498	2.1-437
● 2	1	902	1002	990	3	3	3	639	637	809	9.01k	1.01k-1.02k	1.01k-1.01k	111.4	111.4	111.4	80.9	9.0-9.1	9.0-9.1
● 3	5	2209	7874	4514	17	79	23	2592	5552	3960	5.08k	1.02k-5.07k	1.02k-5.08k	111.4	77.3	84.9	45.6	13.2-65.6	12-59.8
● 4	5	3352	12068	5222	31	152	37	3903	9440	5188	5.08k	1.02k-5.07k	1.02k-5.08k	111.4	73.2	85.0	45.6	13.9-69.3	12-59.8
● 5	11	3768	6154	4072	6	12	6	2172	6418	2422	55.0k	2.51k-66.0k	2.51k-11.0k	42.4	64.7	45.2	1298.7	38.8-1.02k	55.5-243
● 6	1	2272	2453	2579	50	50	50	2236	2154	2797	38.0k	1.01k-1.04k	1.01k-1.04k	111.4	111.4	111.4	341.2	9.1-9.3	9.1-9.4
● 7	303 & 402	306	1250	664	3	9	3	142	1606	519	30.9k	30.4k	30.4k	143.3	61.1	112.8	215.7	498	270
● 8	3	91	785	829	0	0	0	109	643	677	2.00M	1.00M-2.00M	1.00M-2.50M	224.2	64.4	58.6	8.92k	15.5k-31.0k	17.1k-42.6k
● 9	4	1466	6620	3179	14	40	10	1831	5943	3475	10043	1460	1464	120.0	46.1	45.0	83.7	31.7	32.5
● 10	10 & 10 & 10	1746	11068	4807	12	128	28	1784	10509	4481	20065	10896	10899	120.0	78.2	82.6	167.2	139.4	132.0
● 11	4 & 13	2241	4788	2832	5	39	19	1642	4808	3180	95-210	89-105	92-110	120.0	83.8	108.1	1.3	1.1-1.3	0.9-1.0
Normalised geom. mean		1×	**2.5**×	**1.5**×	1×	**2.7**×	**1.1**×	1×	**3.3**×	**1.6**×	1×	**0.3-0.8**×	**0.3-0.6**×	1×	**0.9**×	**0.9**×	1×	**0.5-1**×	**0.3-0.7**×

at (1,1). The starting point of an arrow represents the LUT usage and execution time of the dynamically scheduled hardware, while the corresponding result of the DASS hardware is at the end of the arrow. Here we use η to indicate the fractions of long latency operations. The II of the static island in the DASS hardware is chosen only considering the worst case of the execution patterns, where all the iterations are long, that is $\eta = 1$, assuming the user does not know the input data distribution. With fixed hardware architecture, we show the results of all seven benchmarks with different input data distributions. Generally, our DASS designs sit at the top left of the corresponding statically scheduled hardware. In addition, for the same benchmark, most DASS hardware designs are on the bottom left of the dynamically scheduled hardware. It shows that the DASS hardware can be smaller than the dynamically scheduled hardware and have better performance. The arrows that point to the top right indicate that the benchmark is not suitable for DASS and will be explained in the later section.

The results of those arrows in the figure show different patterns over the performance, attributing to the variety of code patterns in the benchmarks. For instance, some arrows for the same benchmarks position at a noticeable distance from each other, like groups of 3, 4, 5, 8 and 11. The reason is that the performance of the benchmark depends on the distribution of the input data. In contrast, some arrows for the same benchmarks overlap completely, like groups of 7, 9 and 10. These arrows indicate that the input data does not affect the performance of the circuit. Similarly, the benchmarks between these two categories result in the partially overlapping or closely positioned arrows, like groups of 1, 2 and 6. Besides, from the area point of view, the arrows for the same benchmark have the same area reduction as the hardware architecture is fixed.

Detailed results of these benchmarks are shown in Table 3.2, considering all the cases of possible input distribution (from all long to all short). In general, some values in the "Total Cycles" column is represented as a range because the control decisions taken in the code depend on input values. For the statically scheduled case, the number of total cycles is often independent of the input due to pipelining worst-case assumptions made by the static scheduler. However, in the case of 1) `sparseMatrixPower` and 11) `getIntersection`, the static scheduling scheduler decides to implement the outer loop of the circuit without pipelining, resulting in sequential

execution of iteration and hence also variable execution time. There is also a small difference between the total clock cycles of the dynamically scheduled hardware and the DASS hardware. One of the reasons is that the existence of bubbles causes pipeline stalls at startup and then the throughput is stabilised. The cycle count of the statically scheduled hardware in the DASS hardware may also be different from the corresponding dynamically scheduled hardware due to different retiming approaches, which also affects the critical path (like function `g` in Section 3.2.2).

In some of the benchmarks like 3) `filterSum`, the II of the top function is bounded by the topology of the circuit, leading to more area saving. For the benchmarks that contain sparse data operations like 1) `sparseMatrixPower`, although the memory is shared, it can be proved that there is no memory conflict between the static islands and the dynamically scheduled surroundings. Therefore the design quality of the hardware can still be improved by DASS. The dynamic pipelining capabilities are not always as powerful as those of static scheduling when pipelining more complex loops (*i.e.* the dynamically scheduled hardware sometimes contains more restrictive synchronisation logic which may prevent complete loop pipelining). Hence, in 5) `getTanh`, the DASS design benefits in cycle count by introducing the fully pipelined static island. The benchmark 7) `BNNKernel` shows that multiple static islands can be synthesised using our tool. Ideally, all the regular operations in the input code can be synthesised as statically scheduled hardware to maximise area efficiency and performance. Besides, 11) `getIntersection` has both an unbounded loop and the data-dependent `if` conditions that cannot be fully pipelined by static scheduling. However, the program contains several operations that can be shared as the performance bottleneck is at the memory bandwidth. We identify the static islands based on the conditions given in Section 3.2.3, and the tool synthesises these functions from the original program into static islands that both access the same array. The analysis for dynamic scheduling shows that these two functions have no memory conflict, as they never compute in parallel. The average results do not include an unsuitable benchmark 8) `bubbleSort` causing a large bias on the average. Over the 10 benchmarks amenable for DASS, dynamic scheduling achieves 0.03-1.52× of the execution time and 1.07-6.34× area, while DASS achieves 0.04-1.31× of the execution time and 1.09-2.75× area. If the input data distribution is known, the design

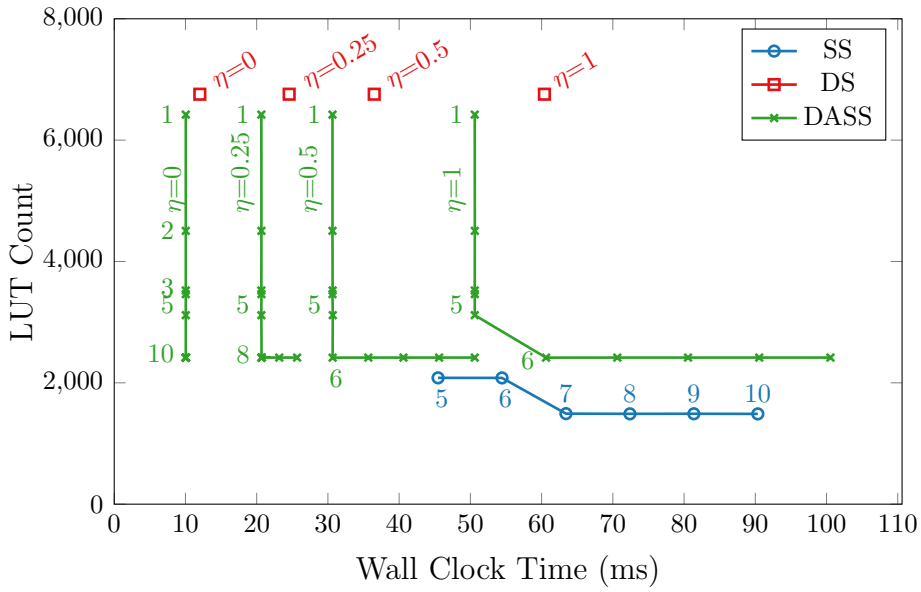


Figure 3.11: LUT usage of different scheduling approaches over the performance for the example from Figure 3.3. Each data point on DASS is labelled with the II of function `g`. Each data point on static scheduling is labelled with the loop II. η indicates the fractions of long latency operations. DS = dynamic scheduling, and SS = static scheduling.

quality of the DASS hardware for all these benchmarks can be further improved (as in case study 1).

3.2.5 Case Study 1: II Exploration

Now let us take the motivating example in Section 3.2.2 for a case study. We discuss how the variation of II affects the hardware performance and some principles that guide the selection of an appropriate II for the static island of a DASS circuit.

The Effects of II Selection

Let us first consider the case when the entire circuit is generated using static scheduling. As an example, in Figure 3.3 the minimum II of the loop in function `filterSum` is 5, because of a loop-carried dependency on `s` that takes 5 cycles. However, a user can also choose a larger II. This can lead to a smaller area (because of more opportunities for resource sharing) but higher latency, as shown in Figure 3.11 (blue circles). In this case, if the II is increased from 5 to 7,

the LUT count is reduced by 28%, at the cost of increasing the latency by 39%.

Now let us consider the DASS case. For the example in Figure 3.3, there are various choices of II for function `g`. The most aggressive solution is to set II = 1 for the highest possible throughput. However, due to the aforementioned loop-carried dependency on `s`, there is actually no performance benefit if the II is set to anything below 5—the loop-carried dependency dictates that the time between two calls to `g` is at least 5 cycles (when `g` is called from two consecutive loop iterations). The time between calls to `g` will only increase if the iterations that call it are further apart. Hence, if the user's primary objective is to maximise performance, an II of 5 cycles is sensible.

However, if the user knows more about the expected data distribution of the input, they can choose an even better II for `g`. Let us explore these effects on the motivating example. The fraction of the iterations where `d < 0` is 1 - η. The input data distribution affects η over all the loop iterations. Figure 3.11 shows the LUT usage and wall clock time of the hardware by three scheduling approaches over several values of η, the fractions of long latency operations. The circuit is buffered for throughput by Dynamatic [110], and we assume the decision of the `if` condition is uniformly distributed over the iterations. Unbalanced structures are usually handled by the buffers, so here we only consider the average case. In the figure, it can be seen that the best II for function `g` varies in terms of the input data distribution. For example, if only the odd loop iterations are long and the rest are short, η is 0.5 and the optimal II for function `g` is 6.

More generally, a suitable II can be selected for a function `f`, where $\frac{1}{II}$ of a component can be considered as its maximum rate of processing data (also known as maximum throughput). The maximum II of function `f` that does not affect the overall program execution time is defined as its *DASS optimal II (II_{opt})*. It depends on two constraints, the maximum production rate allowed by its predecessors, $\frac{1}{II_p}$, and the maximum consumption rate allowed by its successors, $\frac{1}{II_s}$.

II Selection for the Motivating Example

Now let us show how to select the *DASS optimal II (II_{opt})* for the motivating example. Other works have investigated these effects [110, 111, 112] in related problems. Here is an example of how it works. To analyse the circuit, we first show how to formalise the input rates and output rates for each component. Then we show how to use the formulation to find the *DASS optimal II (II_{opt})* for the example.

There are two types of components in the dynamically scheduled hardware: non-control-flow components and control-flow components. The non-control-flow components only perform predictable operations like addition and multiplication. The control-flow components perform unpredictable operations like merge and branch.

For a non-control-flow component g that has N predecessors and M successors, let the actual data rate of its i^{th} predecessor be r_{pi}, the actual data rate of its i^{th} successor be r_{si}. Then we can see that all these rates are equal, where we introduce r to represent the value that these rates are all equal to:

$$\forall i, j \in [1, N],\ r_{pi} = r_{sj} = r \tag{3.23}$$

The reason is that the handshaking interface of the component stalls the inputs until all the inputs are valid and all the predecessors are ready. This balances the rates at the inputs and outputs. Also, the data rate is limited by the physical IIs of the component and all the surrounding hardware.

$$r \leq \min(\frac{1}{II_g}, \frac{1}{II_{p1}}, \ldots, \frac{1}{II_{pN}}, \frac{1}{II_{s1}}, \ldots, \frac{1}{II_{sM}}) \tag{3.24}$$

On the other hand, the processing rates of control flow components depend on the topology

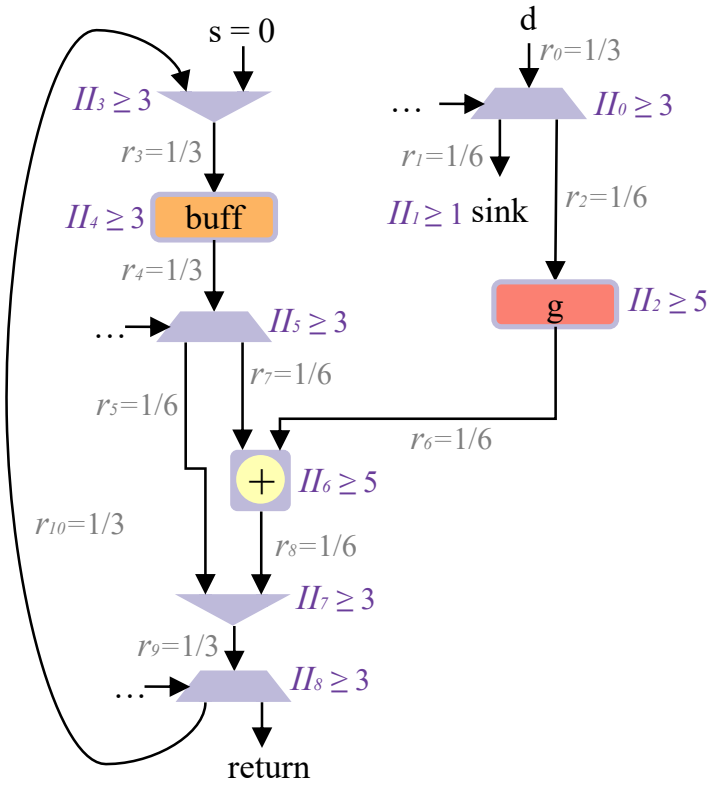


Figure 3.12: Rate analysis for the motivating example.

of the circuit and the input data distribution. The rate changes when the data goes through these components, as detailed below:

$$\text{Merge:} \qquad r_{\text{out}} = r_{\text{in1}} + r_{\text{in2}} \tag{3.25}$$

$$\text{Branch:} \qquad r_{\text{in1}} = r_{\text{in2}} = r_{\text{out1}} + r_{\text{out2}} \tag{3.26}$$

The rate analysis of a portion of the dataflow circuit from Figure 3.4b is shown in Figure 3.12. The green labels show the II constraints of each component and the blue labels represent the actual rate along each edge between two components. Due to the loop-carried dependency on the adder, where the output `s` is sent back to the input, the II of the circuit is limited by the latency of the feedback loop containing an adder and a buffer. That latency is 5 cycles, hence any value of the II of that adder smaller than 5 does not cause a performance bottleneck. In this case, the input and output rate of the adder is limited: $II_6 \geq 5$. Since there is no dataflow

component in the path, $II_2 = II_6 \geq 5$. The top component consuming `d` is a *branch* component, which sends data to one of two outputs according to the `if` condition. The rate of a *branch* component with the loop-carried dependency has an additional constraint:

$$\frac{1}{r_{\text{branch,in1}}} \geq \max(II_{\text{in}},\ II_{\text{out1}} \times p_1 + II_{\text{out2}} \times (1 - p_1)) \tag{3.27}$$

where II_{in} is the II of its predecessor, II_{out1} is the II of its first successor, p_1 is the fraction of the data going into its first successor, and II_{out2} is the II of its second successor. In the figure, the predecessor is known not to be the bottleneck as the upper loop in Figure 3.4b can feed `d` every clock cycle. In addition, one of the successors, *sink*, has $II_1 \geq 1$ as it can take data every clock cycle, and the other is function `g` with $II_2 \geq 5$. With half of the iterations being long, that is $p_1 = 0.5$, we have $II_0 \geq 0.5 \times 1 + 0.5 \times 5 = 3$. This means the highest overall rate is $r_0 = \frac{1}{3}$, where the component consumes 1 set of data every 3 cycles on average. This agrees with the schedule in Figure 3.5b that the hardware consumes 2 sets of data every 6 cycles and repeats. At the highest rate, the rate of the input is split into two edges through the *branch* component in terms of the fraction of the data going into the corresponding successor:

$$r_{\text{out1}} = p_1 \times r_{\text{in}}, \tag{3.28}$$

$$r_{\text{out2}} = (1 - p_1) \times r_{\text{in}} \tag{3.29}$$

In this case, $r_2 = \frac{r_0}{2} = \frac{1}{6}$. Similar analysis can be performed on other *branch* components in the circuit, resulting in the rate of each edge shown in Figure 3.12. In conclusion, the rate to the function `g` is $\frac{1}{6}$ at the highest overall rate, and the *DASS optimal II* of function `g` is $II_{\text{opt}} = 6$. Smaller IIs may cause less area saving and larger IIs may cause performance degrading.

For all input data distributions in Figure 3.11, the static-scheduling approach appears as a single line. The dynamically scheduled solution is always the same hardware architecture (*i.e.* constant LUT count) but with performance varying with the changing input data distribution. Our approach is shown as multiple green lines, one for each input data distribution. The design

with *DASS optimal II*, shown as the elbows in the DASS lines, can have better performance than the dynamically scheduled hardware by improving the maximum clock frequency using static scheduling inside static islands. In addition, the DASS hardware can also have comparable area efficiency compared to the statically scheduled hardware in terms of LUT and DSP usage.

By performing the rate analysis above, we have the *DASS optimal II* of function `g` equal to $II_{\text{opt2}} = \frac{1}{\eta+4}$. This can be justified as follows. Knowing $II_d = 1$, we have $II_0 = (1-\eta) \times II_1 + \eta \times II_2$. Then knowing $II_1 \geq 1$ and $II_2 \geq 5$, we have $II_0 \geq 1 + 4L$. For best performance, $II_0 = 1 + 4\eta$. Ultimately, knowing $r_0 = \frac{1}{II_0}, r_2 = r_0 \times \eta$ and $r_2 = \frac{1}{II_{\text{opt2}}}$, we have $II_{\text{opt2}} = \frac{1}{\eta+4}$, as required.

For instance, at $\eta = 1$, the *DASS optimal II* is 5, the same as the minimum loop II from static scheduling. When $\eta = 0$, function `g` and the adder for `+=` are never used, so the II of the function does not affect the latency of the whole program. In this case, the *DASS optimal II* is infinity, *i.e.* function `g` is no longer needed.

In this work, we let users manually determine the optimal II for the static island in the DASS hardware. In general, finding the optimal II for a static island can be difficult as it depends on both the topology of the circuit and the input data distribution. However, even if users have only some information on the circuit, such as the minimum II achievable due to loop-carried dependencies, the hardware optimisation is still promising. In the figure, the DASS hardware with $II = 5$ for the static island does not have minimum area but still achieves significant area reduction compared to the dynamically scheduled hardware. Although the hardware is underperforming, the difference in area reduction by switching from $II = 5$ to $II = 6$ is significantly smaller than that by switching from $II = 1$ to $II = 2$.

3.2.6 Case Study 2: BubbleSort

There are two throughput overheads caused by DASS. These overheads are usually minor but can still affect the design quality with a bad choice of source for static islands. Here we take `bubbleSort` for example, as illustrated in Figure 3.13. The swapping process is chosen to be

```
1 void bubbleSort(int A[N]){
2   for (int i = 0; i < N-1; i++){
3     for(int j = i; j < N-1-i; j++){
4       /*if (A[j] < A[j+1]) {
5         A[j] = A[j+1];
6         A[j+1] = A[j];
7       }*/
8       swap_ss(A, j);
9 }}}
```

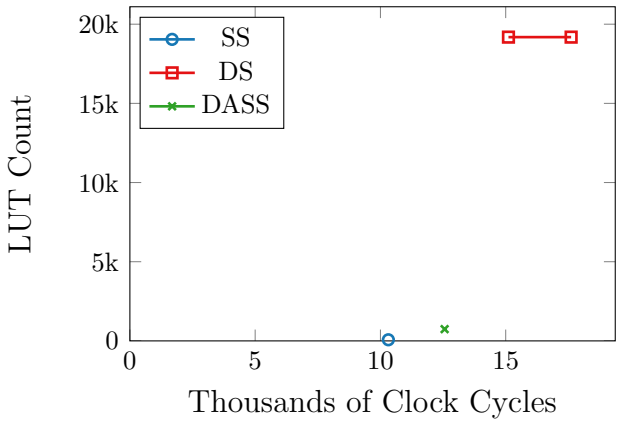


Figure 3.13: Design quality of different scheduling approaches for `bubbleSort` benchmark. Bad choices may result in worse performance and a larger area. DS = dynamic scheduling, and SS = static scheduling.

a static island named `swap_ss`. The first overhead is caused by the inter-iteration dependence inside `swap_ss`. A load from `A[j]` in an iteration depends on the conditional store to `A[j+1]` in the last iteration. This memory dependence forces the schedule `swap_ss` to be sequential to preserve correctness. Second, the extraction of static islands causes one cycle of additional latency. This additional latency is usually hidden in a pipeline. In `bubbleSort`, the sequential schedule of `swap_ss` causes throughput of the loop depending on the latency of `swap_ss`, slowing down the computation.

The bottom of Figure 3.13 shows how these overheads affect the performance. In the figure, it can be seen that the optimal choice for `bubbleSort` is to statically schedule the whole program. Compared to the statically scheduled hardware, the DASS hardware has lower throughput due to the additional latency caused by the wrapper. It also loses in the circuit area due to the handshaking interface in the dynamically scheduled hardware. In addition, the dynamically scheduled hardware can solve the memory dependence using an LSQ. However, the LSQ has a large area and a long memory latency. The memory latency is usually hidden in the pipeline but not for `bubbleSort` as explained before. Such a long latency results in a low throughput in

the dynamically scheduled hardware. Compared to the dynamically scheduled hardware, the DASS hardware does not have an LSQ and has a smaller memory latency since all the memory accesses are statically scheduled.

3.3 Summary

In high-level synthesis, dynamic scheduling is useful for handling irregular and control-dominated applications. On the other hand, static scheduling can benefit from powerful optimisations to minimise the critical path and resource requirements of the resulting circuit.

In this chapter, we formalise existing dynamic scheduling and static scheduling for HLS tools, and identify the opportunities for achieving a better scheduling solution by an optimised formulation. We combine existing dynamic and static HLS approaches in our DASS HLS tool flow to strategically replace regions of dynamically scheduled hardware with their statically scheduled equivalents: we benefit from the flexibility of dynamic scheduling to achieve high throughput as well as the frequency and resource optimisation capabilities of static scheduling to achieve fast and area-efficient designs.

Across a range of benchmark programs that are amenable to DASS, our approach achieves 0.43-1.05× the area of the corresponding dynamically scheduled design, and results in 0.76-25× the speed of the corresponding statically scheduled design. In certain cases, the knowledge of the input data distribution allows us to further increase the design quality and may result in additional performance and area improvements. Our current approach relies on the user to annotate via pragmas parts of the code which do not benefit from dynamic scheduling and can, therefore, be replaced with static functions. In the next chapter, we will introduce how to explore the automated recognition of such code and these pragmas.

Chapter 4

Finding and Balancing Static Islands

This chapter combines the publication at FPGA 2022 and FCCM 2021. It tackles the challenges of automating the process for scheduling optimisation for DASS hardware. First, we explain the work published at FPGA 2022 and show how a set of good static islands should be determined from arbitrary programs. Second, we extend the publication at FCCM 2021 by adding continuous Petri net formulation and show how these static islands should be optimised by rate balancing with their dynamically scheduled surroundings.

4.1 Finding Static Islands

In this section, we demonstrate how to solve challenge 2 in Section 3.1.5 for automatically determining good static islands from arbitrary programs. A good static island should achieve significant resource sharing inside the island with a minimal impact on the overall performance.

In Section 4.1.1, we explain the motivation and challenges of finding good static islands from arbitrary programs. In Section 4.1.2, we give a motivating example to illustrate the challenge in selecting scheduling approaches. In Section 4.1.3, we present related works and compare them with our approach. In Section 4.1.4, we formalise the problem and present a tool to automatically find good static islands. In Section 4.1.5, we evaluate the effectiveness of our tool on a set of benchmarks.

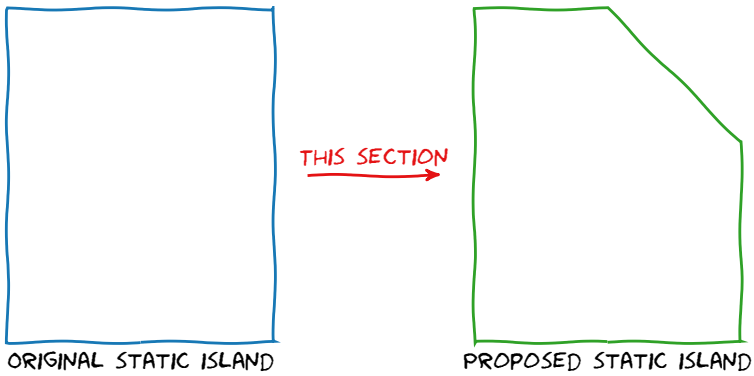


Figure 4.1: A sketch of derectangularising a static island.

4.1.1 Introduction

High-level synthesis today has become popular for designing large and complex hardware designs. Still, it remains the case that automatically synthesising high-performance and area-efficient hardware from arbitrary high-level programs is challenging. To tackle this problem, techniques have been proposed to produce optimised hardware architectures for specific domains, such as image processing [98, 113], deep learning [97, 114] and other matrix computation-based applications [115]. However, users of these tools are still required to have hardware background to write efficient code for optimal performance. The problem remains unsolved for general applications due to the presence of complex control flow that can only be partially parallelised [116].

As explained in Section 3.2, DASS combines the best of both worlds by supporting statically scheduled hardware that forms internal components, named *static islands*, within dynamically scheduled hardware. However, this work left it to the user to determine which pieces of code in an arbitrary program should form the static islands and which parts should be dynamically scheduled. The primary contribution of this section, then, is a heuristic-driven technique for finding good static islands in arbitrary code.

Our secondary contribution has to do with how these static islands, once identified, are interfaced with the surrounding dynamically scheduled hardware. Prior work assumes that when a static island has multiple inputs, all inputs are required at the start of computation [1]. This assumption can lead to huge inefficiency in the final circuit, because if some inputs are valid

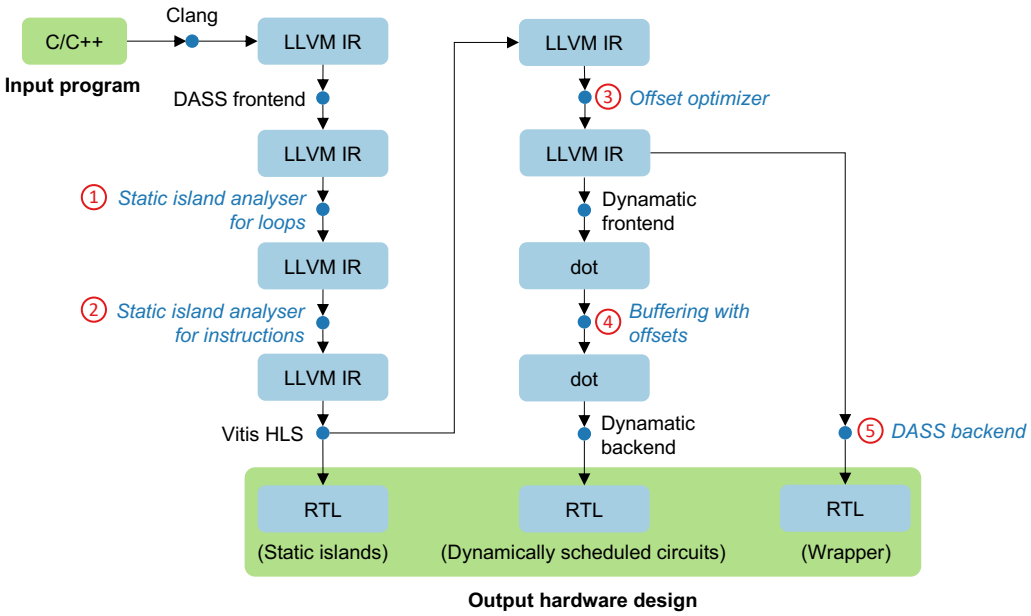


Figure 4.2: Our work integrated into the DASS tool flow. The steps numbered ① to ⑤ are contributions of this section. Section 4.1.4 explains the details.

before the others, they must be held until the others are valid too, even if some of the static island's computation could proceed using only those inputs that are valid. We remove this assumption made in [1], and demonstrate that doing so can lead to an up to 2.6× speedup. As an analogy, we can imagine a timing diagram with time on the vertical axis and resources on the horizontal axis. In such a picture, the static islands introduced in [1] can be thought of as *rectangles* (where all the inputs are aligned on the top edge). Meanwhile, the static islands in this work have a less regular shape. We refer to the process of 'derectangularising' the islands, as shown in Figure 4.1, because it is our method for squeezing better performance out of them.

Taken together, our two contributions extend the state of the art in HLS by automating the choice between static and dynamic scheduling and efficiently synthesising hardware that combines both scheduling approaches. In summary, this section presents:

- a technique that finds an optimised allocation of static islands by analysing the features of each code region and evaluating the hardware performance by different scheduling strategies (① and ② in Figure 4.2),
- an automated HLS pass that calls the commercial tool Xilinx Vivado HLS to efficiently synthesise static islands with high performance by supporting non-simultaneous inputs

```
1  float a[N], r[N];
2  void vecNormTrans() {
3    float weight = 0.0f;
4    loop_0:
5    for (int i=0; i<N; i++) {
6      float d = a[i];
7      if (d < 1.0f)
8        // weight = ss_func_0(d, weight);
9        weight = ((d*d+19.5)*d+3.7)*d+0.73*weight;
10   }
11   // ss_func_1(weight);
12   loop_1:
13   for (int i=0; i<N-4; i++)
14     r[i+4] = r[i]+a[i]/weight;
15 }
```

Figure 4.3: Motivating example for finding static islands.

(③, ④ and ⑤ in Figure 4.2), and

- analysis and results showing that the proposed approach achieves 0.13-1.05× the area combined with 0.75-1.61× the performance through automatic identification and synthesis of static islands compared to fully dynamically scheduled circuits. Compared to the fastest designs discovered via exhaustive search, our approach generates designs that are within 2% of the performance.

4.1.2 Motivating Example

In this section, we use a motivating example to demonstrate the challenge in selecting code regions for static scheduling within dynamic surroundings. Figure 4.3 shows a code example to be scheduled and synthesised. In the code, a loop named `loop_0` loads an array element `a[i]` and adds a Horner-style polynomial [117, 118] evaluated at `a[i]` onto a variable named `weight` if the array element is less than 1. The value of `weight` is then used in a subsequent loop named `loop_1` for the transformation of array `r`.

Figure 4.4 shows the corresponding data flow graph assuming it is fully dynamically scheduled using the approach presented in [13]. DASS in Chapter 3 statically schedules part of the data flow graph as individual static islands, indicated as dotted circles, for resource sharing inside each island. Combining dynamic and static scheduling techniques onto a single circuit can achieve both high performance and area efficiency. Dynamic scheduling is beneficial for input-

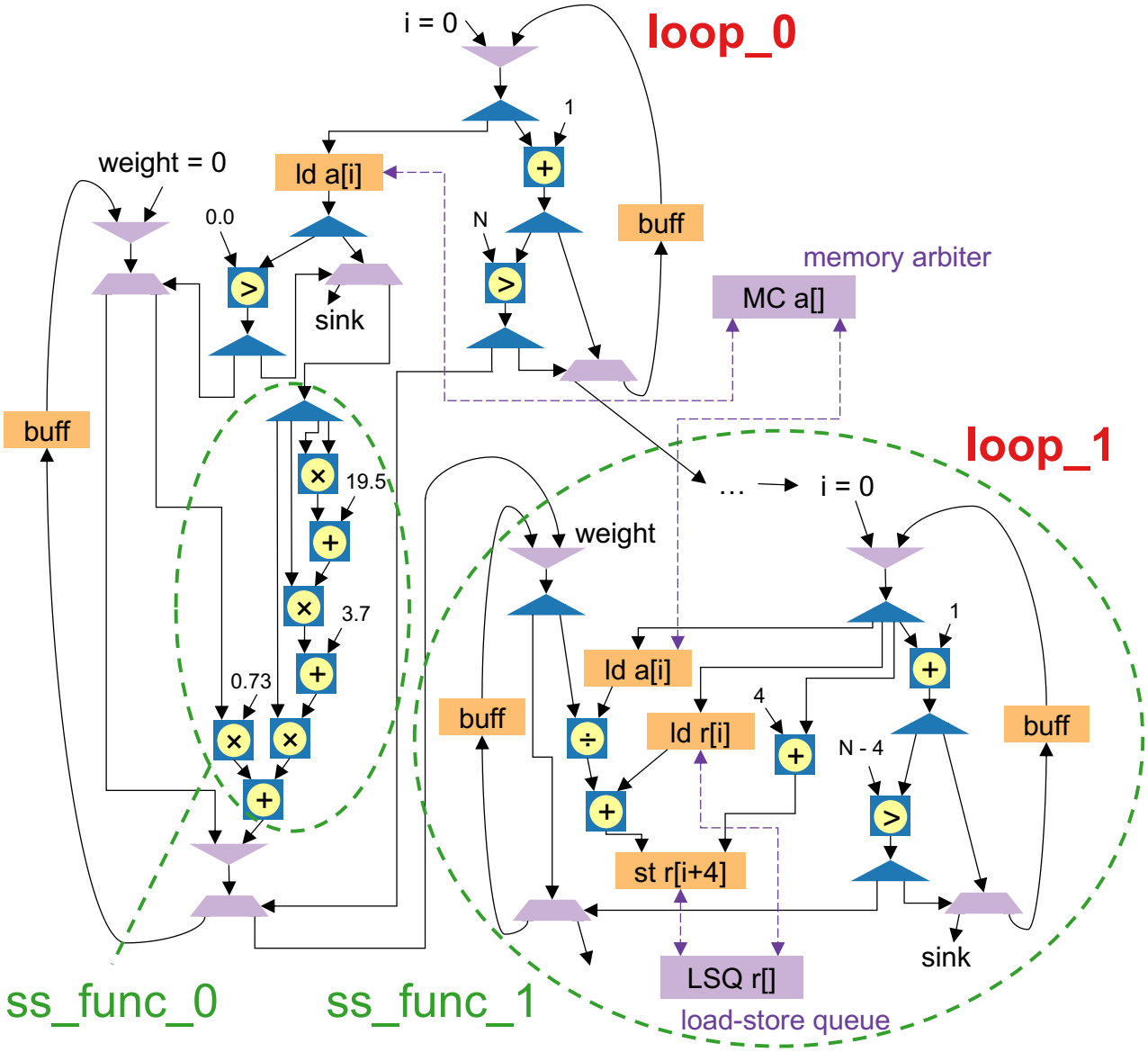


Figure 4.4: Data flow graph from the example in Figure 4.3. The polynomial expression on line 9 and the loop named `loop_1` should be synthesised as two static islands.

Table 4.1: Comparison between our approach and other approaches using static scheduling or dynamic scheduling only.

	LUTs	**DSPs**	**Cycles**	**Fmax (MHz)**	**Time (μs)**
Vivado HLS (conservative II)	1614	5	12327	120	102
Dynamatic (optimistic II)	21290	20	6079	90	68
Our work	2860	7	6703	106	63

dependent control flows, such as the `if` condition in `loop_0`, while static scheduling is beneficial for predictable data flow, such as the polynomial expression on line 10. The goal of our work is to determine all the best possible static islands under given performance constraints and efficiently synthesise them.

Automatically determining static islands is challenging. In Figure 4.5, we enumerate all the possible sets of (non-overlapping) subgraphs (each of which must have at least two instructions to be worth statically scheduling) from the graph in Figure 4.4, resulting in 102 designs. The design that only uses dynamic scheduling is at the bottom right of the figure; it has low latency but a large area. The design that only uses static scheduling is at the left of the figure; it has higher latency but a smaller area. The other designs use both approaches, with different code regions using different approaches. Some designs perform much worse than either fully static or fully dynamic scheduling. For instance, the design at the top right of the figure has high latency and a large area. This motivates us to formalise and automate the search for code features that are amenable to static scheduling as part of a fully-automated tool flow.

For this example, our tool suggests statically scheduling the polynomial expression on line 10 and the entire `loop_1` as shown in Figure 4.4. This results in a design with similar performance to the dynamically scheduled design but a similar area to the statically scheduled design, as indicated in Figure 4.5. The absolute results are shown in Table 4.1.

The selection suggested by our tool results in a high-quality design for three reasons. First, the synthesised design still competes at a high throughput because the input-dependent `if` statement remains dynamically scheduled. Second, DSPs are significantly reduced due to resource sharing in the polynomial expression. Finally, instead of using area-expensive load-store queues (LSQs) to handle inter-iteration memory dependence for dynamic scheduling, statically

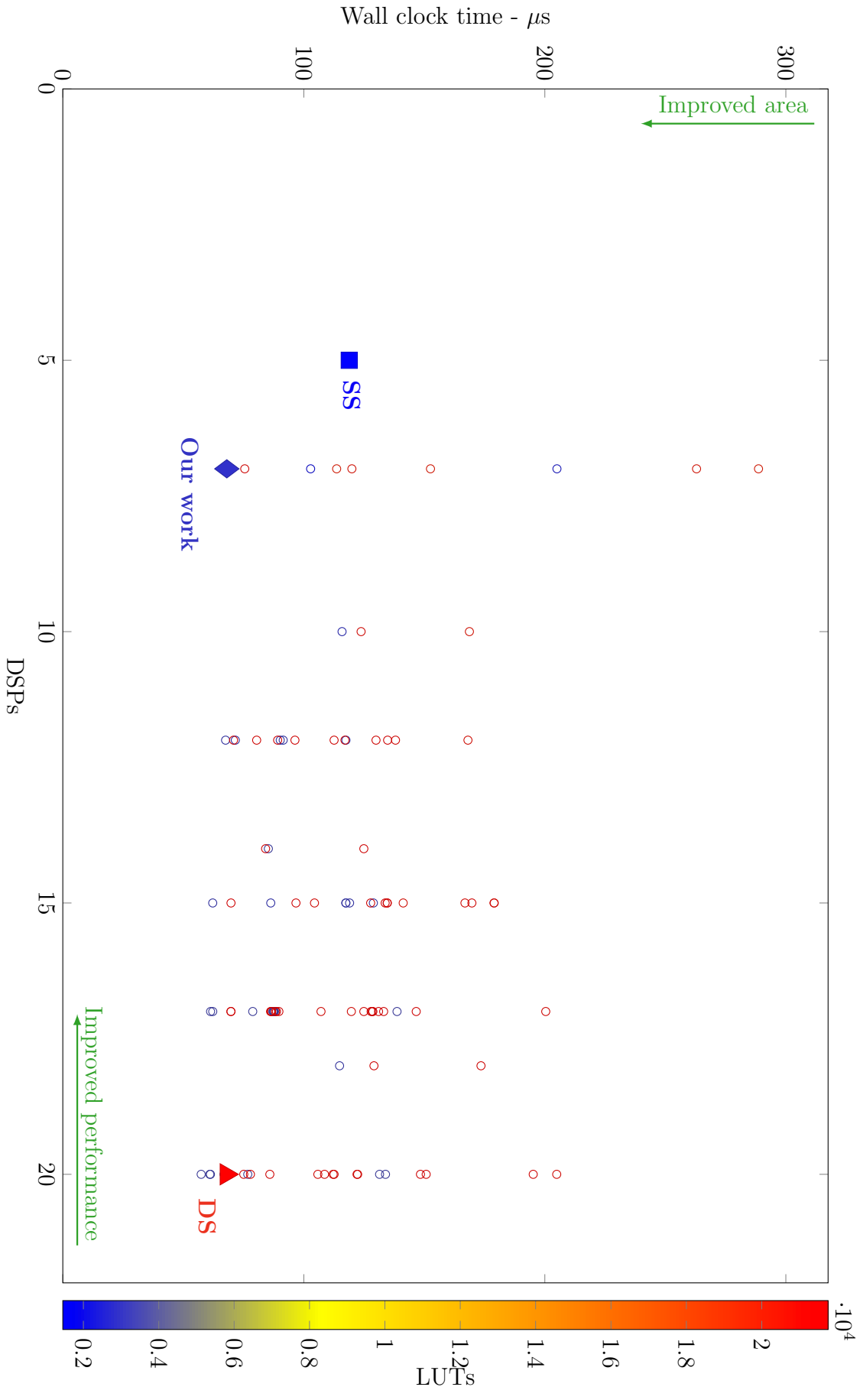


Figure 4.5: Area and delay of designs for all possible selection of static islands. There are 102 possible selections of static islands, and most of them lie at the top right of the figure (big & slow). Automatically determining the optimal selection is challenging.

scheduling `loop_1` saves many LUTs and still achieves the same throughput. In the rest of this section, we will explain in detail how these features are modelled and optimised by our tool.

4.1.3 Background

Module selection is a process to select an optimal module design among a set of choices with the same functionality to improve performance or area. Module selection in HLS has been widely studied. Ishikawa and De Micheli [119] propose a module selection algorithm that optimises the schedules of hardware with a finite set of predefined components. Ahmad *et al.* [120] present a problem-space genetic algorithm for static scheduling. Ito *et al.* [121] propose an ILP-based model for optimising the schedule of data flow architecture. Sun *et al.* [122] combine the module selection and resource sharing in design exploration. Cong *et al.* [123] propose an ILP-based scheduling including module selection for streaming applications. However, these approaches all target statically scheduled hardware only. The behaviour of dynamically scheduled surroundings can be unpredictable, and these methods cannot be applied without assuming the worst-case computation.

In dynamic scheduling, latency-insensitive system graphs (lis-graphs) are used for hardware optimisation, such as loop pipelining, re-timing and buffering [124, 125, 126, 127]. This is extended to marked graphs in HLS tools like Dynamatic [13]. However, none of these works optimises the offsets of component ports. Our work optimises the offsets of static islands to achieve better performance (explained in Section 4.1.4).

4.1.4 Methodology

This section shows how to determine good static islands for minimal performance loss and maximal resource sharing. Resources in a static island can only be shared inside the static island due to dynamic behaviour in its surroundings. The scheduler can determine the states of a static island once the island starts to compute, but it cannot determine when it starts in relation to other static islands. Therefore, a larger static island contains more resources to

share and can achieve higher area efficiency. However, if a large static island contains data-dependent operations, the conservatism in static scheduling may cause performance loss and violate performance constraints.

In order to automatically determine these optimally sized static islands, we first summarise the fundamental features of code that are amenable to static scheduling. Based on these features, we then show how our tool extracts static islands. Next, we illustrate how to optimise the interface between static islands and their dynamically scheduled surroundings for high performance. Finally, we demonstrate the complete tool flow integrated into DASS.

Features Amenable for Static Scheduling

In general, static scheduling is optimal if the code behaviour is fully predictable, otherwise, the scheduler always assumes the worst case in time. A code region that is amenable to static scheduling should satisfy the following conditions:

1. Code should have no data-dependent control flow, or only have data-dependent control flow where all control flow paths have the same throughput.

2. If code contains loops, each loop should have no inter-iteration dependence, or only have inter-iteration dependences with constant dependence distances.

For these code regions, the timing of the synthesised hardware is predictable in clock cycles, that is, the statically scheduled design can achieve the same throughput as the dynamically scheduled design. The features that are amenable for static scheduling by Vivado HLS are explained on the next page.

Constructing Static Islands

Constructing static islands is a two-step process, one targeting high-level operations in loops and the other targeting low-level operations in instructions. For each function, if the function

contains loops, our analyser first checks whether each loop can be statically scheduled. If it can, our tool identifies the whole loop as a static island. Otherwise, our analyser checks instructions in the function/loop body and constructs static islands in the form of groups of instructions. Each group of instructions forms a subgraph in the data flow graph of the input program as illustrated in Figure 4.4.

Step 1: Constructing Static Islands from Loops

Here we introduce four pre-conditions for determining whether a loop should be statically or dynamically scheduled. The first two pre-conditions are restricted by the tools we use, and the other two pre-conditions are restricted by the input code.

Condition 1: Vivado HLS Loop Restrictions. Apart from the features mentioned above, Vivado HLS, the tool we use for static scheduling, has additional restrictions for loops to achieve efficient loop pipelining [128, 129]. First, loop nests that cannot be merged into a single loop cannot be optimally pipelined by Vivado HLS. If a loop nest cannot be merged, Vivado HLS either pipelines the top-level loop with all the inner loops fully unrolled, or only pipelines the innermost loops with the rest of the code sequential. Additionally, support for estimating loop trip count from variable loop bounds is limited in Vivado HLS. The scheduler prefers that all the loop bounds and step of a loop are constant. Finally, all the array indices should be in affine form as the analyser only supports affine analysis. We formalise these features into following constraints:

- L_{single}: the loop is a single loop or the loop is a loop nest that can be merged into a single loop. An innermost loop of a loop nest can be considered a single loop.

- B_{cst}: all the loop bounds and steps are constant.

- A_{affine}: all the array indices are in affine form.

A pre-condition of a loop to be scheduled by Vivado HLS is:

$$C_{\text{VHLS}} = L_{\text{single}} \wedge B_{\text{cst}} \wedge A_{\text{affine}} \tag{4.1}$$

Condition 2: DASS Shared Memory Restrictions. DASS, the tool we use as an HLS tool, also has restrictions on shared memory between static islands and dynamically scheduled hardware. Static islands are treated as black boxes by the dynamically scheduled hardware, and memory dependence between static islands and dynamically scheduled hardware cannot be handled by DASS [1]. That is, a loop should not have memory dependence on external code. In the data flow graph, the memory dependence is controlled by LSQs. Here we restrict that a loop sharing an LSQ with other loops cannot be statically scheduled. We define the following:

- $L = \{l_0, l_1, ...\}$: the set of all the loops in the program.
- Q: the set of all the LSQs in the data flow graph of a program if the program is dynamically scheduled.
- Q_l: the set of all the LSQs connected to the data flow graph of loop l if the loop is dynamically scheduled. $Q_l \subseteq Q$

The restriction by DASS for a loop l is then:

$$C_{\text{DASS}} = (\forall l' \in L \setminus \{l\}.\, Q_l \cap Q_{l'} = \emptyset) \tag{4.2}$$

Condition 3: Loop with No Branches. Now we discuss the conditions restricted by the code. A minimum II for a loop iteration is the number of cycles that the iteration must wait after its last iteration starts. Dynamic scheduling supports different minimum IIs across loop iterations, while static scheduling allows a constant II value for all the iterations. If the minimum II is the same across all the loop iterations, the loop should be statically scheduled. Otherwise, all the IIs are relaxed to the maximum value among all the minimum IIs, which causes reduced throughput.

As explained in Chapter 3, an II of a loop depends on two constraints, iteration latency and inter-iteration dependence [47]. Assuming that all operators take constant time, we define the following constraints:

- D_{inter}: the loop has inter-iteration dependence.
- D_{cst}: the distances of all the inter-iteration dependences in the loop are constant.
- B_{br}: the loop body has branches.

Then the loops that satisfy the following condition should be statically scheduled, where iteration latency and dependence distance are both constant, resulting in a constant II:

$$C_0 = \neg B_{\text{br}} \wedge (D_{\text{cst}} \vee \neg D_{\text{inter}}) \tag{4.3}$$

For instance, a loop below has no branch but inter-iteration dependence on array `A`. When `f(i) = i*i`, the dependence distance varies over iterations, then the loop cannot be statically scheduled. When `f(i) = i+10`, the dependence distance is 10, then the loop can be statically scheduled.

```
1 for (int i=0; i<N; i++)
2   A[f(i)] = A[i];
```

Condition 4: Loop with Branches. A loop that contains branches may have a set of iteration latencies and could lead to a set of IIs. In static scheduling, the maximum value of II among these IIs is used for all the loop iterations. Such approximation could cause throughput loss compared to dynamically scheduling the loop. In order to evaluate the throughput loss caused by the II approximation, we propose a performance model specified by the following constraints:

- $V_C = \{v_0, v_1, ...\}$: the set of all the variables in the loop that have loop-carried dependences.

```
1  float s = 0;
2  float d1, d2;
3  for (int i=0; i<N;
4    i++) {
5    float d = A[i];
6    // P(true)=0.5
7    if (d < 1)
8      // II=9
9      s = s*d + d2*(d+0.1);
10   else
11     // II=5
12     s = s+d1;
13 }
```

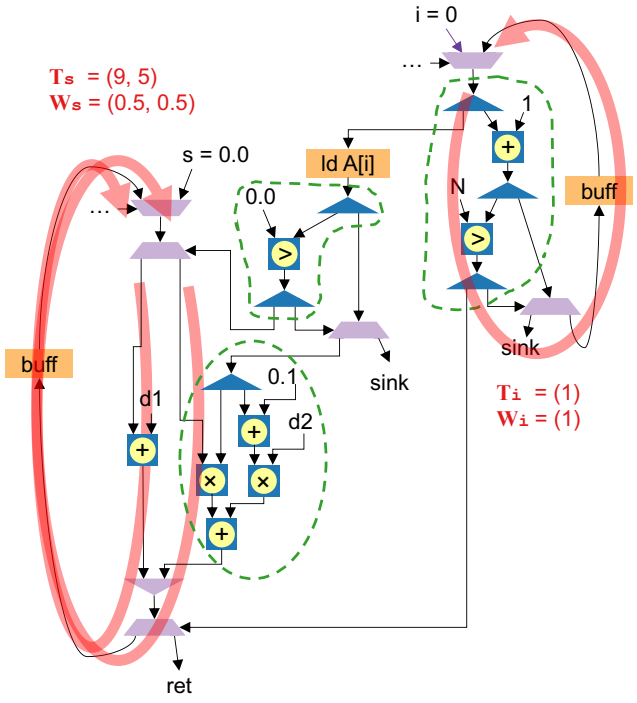


Figure 4.6: A loop example that conditionally updates a variable `s` based on the value of array element `A[i]`. A data flow graph of the loop is shown at the bottom of the figure. The cycles indicate dependences. The loop-carried dependence on `s` has two possible latencies, illustrating two possible IIs. The extracted static islands are shown as dotted circles based on the given $\mathbf{T}_s$ and $\mathbf{W}_s$ in the figure.

- $\mathbf{P}_v$: the vector of all the cycles for a variable that has loop-carried dependences. $v \in V_C$.
- $\mathbf{T}_v$: the vector of the latencies of these cycles in clock cycles.
- $\mathbf{W}_v$: the vector of the probabilities of computing these cycles. $\mathbf{W}_v \cdot \mathbf{1} = 1$.
- λ_0: affordable throughput loss factor.

In order to perform throughput analysis, a data flow graph is generated from the source of the loop. In the graph, each loop-carried dependence on a variable is represented as a set of cycles. For instance, the left side of Figure 4.6 shows a loop example that contains a branch. The loop checks whether the value of an array element `A[i]` is less than 1. If it is, the variable `s` is updated by `s*d+d2*(d+0.1)`, otherwise `s` accumulates `d1`.

The data flow graph of the loop example is on the right of Figure 4.6. There are two variables

that have loop-carried dependences, $V_C = \{\mathtt{s}, \mathtt{i}\}$. In the figure, the cycle on the top right represents the loop-carried dependence on `i`. The value of `i` always increments by 1 in each iteration, where $|\mathbf{P}_\mathtt{i}| = 1$. The cycles on the left represent the loop-carried dependences on `s`. The value of `s` depends on the `if` condition, so there are two cycles in the graph representing the results from true and false branches respectively, where $|\mathbf{P}_\mathtt{s}| = 2$.

Each cycle has a latency and a probability, formalised as elements in $\mathbf{T_v}$ and $\mathbf{W_v}$ respectively. The latency indicates the minimum time in clock cycles required to update the variable with loop-carried dependence, also known as the iteration latency of the cycle. $\mathbf{W_v}$ are obtained by profiling. In Figure 4.6, the cycle for `i` has a fixed latency of 1. Therefore, $\mathbf{T}_\mathtt{i} = (1)$ and $\mathbf{W}_\mathtt{i} = (1)$. Assuming that the probability of the `if` condition being true is 0.5, $\mathbf{W}_\mathtt{s} = (0.5, 0.5)$, and assuming that the latency of a floating point adder is always 5 cycles and the latency of a floating point multiplier is always 4 cycles, the cycle for the true branch has a latency of 9, and the cycle for the false branch has a latency of 5, that is, $\mathbf{T}_\mathtt{s} = (9, 5)$.

The performance of the loop can be estimated using the latencies and probabilities of cycles. Dynamic scheduling supports a variable iteration latency, and the average iteration latency for a variable v is represented as $T_{v,\text{dynamic}}$. Static scheduling supports a constant iteration latency, and the iteration latency for a variable v is then the maximum latency among all the cycles, represented as $T_{v,\text{static}}$.

$$T_{v,\text{dynamic}} = \mathbf{T}_v \cdot \mathbf{W}_v \tag{4.4}$$

$$T_{v,\text{static}} = \max \mathbf{T}_v \tag{4.5}$$

We simplify our analysis by restricting that the loop only has no inter-iteration dependence or inter-iteration dependences with constant dependence distance. A loop that has dynamic dependence distances should be dynamically scheduled. The average IIs are:

$$T_{\text{dynamic}} = \max_{v \in V_C} T_{v,\text{dynamic}} \qquad II_{\text{dynamic}} = \frac{T_{\text{dynamic}}}{\mathrm{d}} \tag{4.6}$$

$$T_{\text{static}} = \max_{v \in V_C} T_{v,\text{static}} \qquad II_{\text{static}} = \frac{T_{\text{static}}}{\mathrm{d}} \tag{4.7}$$

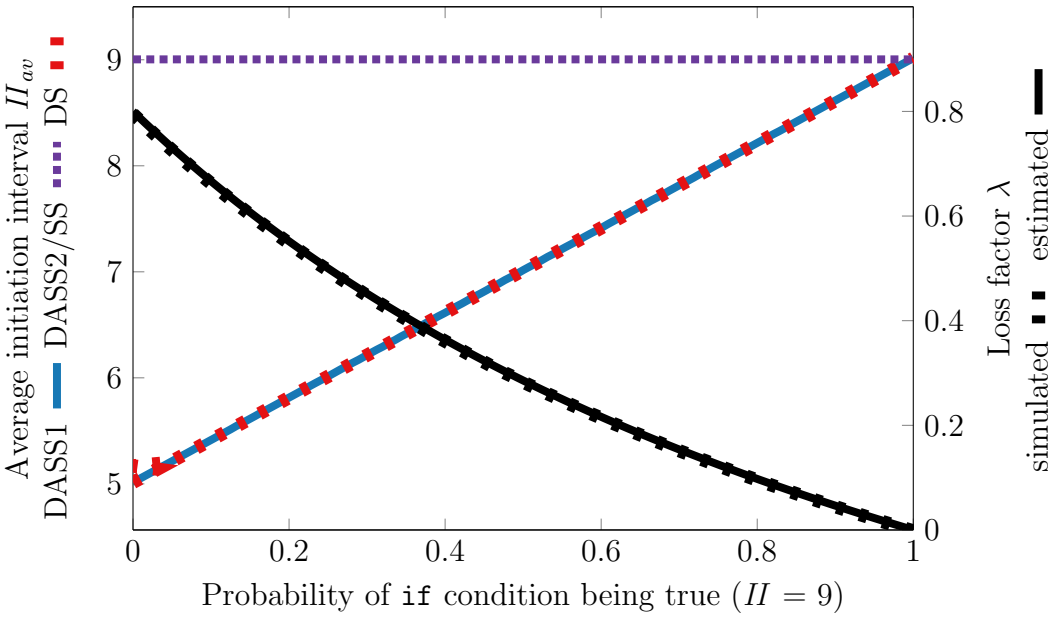


Figure 4.7: DS = the design without any static island. DASS1 = the design statically scheduling line 9 of Figure 4.6. DASS2/SS = the design statically scheduling the whole loop. The average IIs of DASS1 design and DS design vary from 5 to 9 because of the dynamically scheduled `if` condition. The II of DASS2/SS design has a constant II of 9. The loss factor of the loop in Figure 4.6 increases with the probability of the `if` condition being true. The maximum error between the estimated loss factor and the actual loss factor by simulation is 0.5%.

The satisfied loop then has a constant minimum dependence distance, d. The final II of both statically scheduled and dynamically scheduled loops is restricted by the maximum II among all the variables that have loop-carried dependences. In Figure 4.6, the II is restricted by the variable `s`. The performance loss caused by switching a dynamically scheduled loop to a statically scheduled loop can be estimated using a *loss factor* λ.

$$\lambda = \frac{II_{\text{static}} - II_{\text{dynamic}}}{II_{\text{dynamic}}} = \frac{T_{\text{static}} - T_{\text{dynamic}}}{T_{\text{dynamic}}} \tag{4.8}$$

The constant d is then cancelled. If the throughput loss caused by approximating the II is affordable for a user-defined threshold λ_0, the loop is then statically scheduled.

$$C_\lambda = (\lambda_0 - \lambda \geq 0) \tag{4.9}$$

A curve of the loss factor for the loop in Figure 4.6 over the distribution of `if` condition being true is shown in Figure 4.7. Our estimated loss factor is close to the one by simulation. For

Table 4.2: Reference for the scheduling approach that needs to be taken for a loop. DS = dynamic scheduling. SS = static scheduling. DS/SS = depending on our performance model.

Iteration latency	**Dependence distance**		
	No dependence	Constant	Variable
Constant	SS	SS	DS
Variable	SS/DS	SS/DS	DS

the example in Figure 4.6, $\lambda = 0.28$. Assuming $\lambda_0 = 0.05$, our tool suggests to dynamically schedule the loop. The loop still has statically scheduled parts in the loop body, resulting in the DASS1 design by step 2, which are explained later in this section. Based on the performance model, loops that satisfy the following condition should be statically scheduled:

$$C_1 = B_{\text{br}} \wedge (D_{\text{cst}} \vee \neg D_{\text{inter}}) \wedge C_\lambda \tag{4.10}$$

Summarised Condition. The summarised condition that indicates whether a loop should be statically scheduled is shown in Equation 4.11. Table 4.2 summarises Equation 4.11 based on the source patterns. Each loop to be statically scheduled is considered a single static island. Any two adjacent statically scheduled loops are further merged into a static island in step 2.

$$C_{\text{static loop}} = C_{\text{VHLS}} \wedge C_{\text{DASS}} \wedge (C_0 \vee C_1) \tag{4.11}$$

Step 2: Constructing Static Islands from Instructions

If a loop is not amenable to static scheduling, our tool extracts static islands from the loop body at the instruction level. We consider each extracted statically scheduled loop or each dynamically scheduled operation as a single node in the data flow graph of the function. We define the following terms:

- N: the set of all the nodes in the data flow graph.
- $E \rightarrow N \times N$: the set of all the edges in the graph.
- $M \rightarrow \{0, 1\}$: whether these nodes can be merged. 0 = no, 1 = yes.

Table 4.3: Static islands found over a set of benchmarks.

Benchmarks	# Islands	Benchmarks	# Islands
vecNormTrans	2	getTanh	1
doitgenTriple	2	covariance	3
correlation	5	syr2k	1
levmarq	2	gemver	3
gramSchmidt	7	gesummv	2

- $S \to \mathbb{N}$: the ID of the static island to which a node belongs.

We specify the merging rule as follows:

1. Each node is considered a static island.
2. A node that does not have the same throughput at all its input and output ports cannot be merged, *e.g.* *Merge*, *Branch* and *Mux*. This avoids performance loss in pipelining data-dependent operations.
3. Nodes that connect to an LSQ cannot be merged for the same reason explained in Condition 2.

Based on the rules above, our tool iteratively merges nodes into larger static islands. For instance, the dotted circles in Figure 4.6 represent the merged static islands in the graph. Each island can be synthesised into a single component. The following condition holds after extracting static islands.

$$\forall (n_0, n_1) \in E, S(n_0) = S(n_1) \vee \neg(M(n_0) \wedge M(n_1)) \tag{4.12}$$

The main benefit of static scheduling is enabling resource sharing in a static island. As an optimisation process, our tool evaluates the size of each island by counting the number of operations. If an island is too small that has no space for resource sharing, then it remains dynamically scheduled. For the example in Figure 4.6, only the island at the bottom left is amenable to static scheduling, while the other two islands are ignored. Table 4.3 shows the number of static islands found over a set of benchmarks.

Optimising Static Islands using Offsets

Once a static island is determined, it is synthesised and placed in a wrapper to communicate with its dynamically scheduled surroundings. The wrapper monitors and controls the computation of the static island based on its input and output states.

An offset of an input means the number of cycles between the start time of the computation and the time when this input is firstly used. An offset of an output means the number of cycles between the end time of the computation and the time when this output starts to be valid. This section introduces how to use the offsets to optimise the wrapper to improve the overall throughput. We first compare the throughput of the original wrapper in DASS and our proposed wrapper. Then we show how the proposed wrapper is implemented and its additional constraints.

Throughput of Original DASS Wrapper Design

The offset can be used to determine when an input/output is required, which can affect the overall performance. The original wrapper assumes zero offsets for all the inputs, which requires all the inputs to be valid to start the computation [1]. When a static island is in a loop and has a loop-carried dependence, the latency of the static island affects the overall throughput. For a static island, let M be the set of its inputs and N be the set of its outputs. The loop-carried dependence set among these inputs and outputs over all the iterations is D, where:

$$D \subseteq M \times N \times \mathbb{N} \times \mathbb{N} : (m, n, k_1, k_2) \in D \tag{4.13}$$

k_1 and k_2 are the iteration indices, where $k_1 > k_2$. For instance, (m, n, k_1, k_2) means the input m in iteration k_1 depends on the output n in iteration k_2.

Now we formalise the time constraints. All the times are in clock cycles. Let $t_{m,k}$ be the time when input m in iteration k is *consumed* and $t_{n,k}$ be the time when output n in iteration k

becomes valid.[1]. The dependence constraint can be formulated as follows:

$$\forall m, n, k_1, k_2.\, (m, n, k_1, k_2) \in D \Rightarrow t_{m,k_1} \geq t_{n,k_2} \tag{4.14}$$

The dynamically scheduled surroundings use a handshake interface to ensure that Equation 4.14 *always* holds. Here we use Equation 4.14 as a pre-condition for the following analysis in the static island.

Assume that a static island is pipelined with a constant II and always has the same latency. Let α_m be the offset of input m and L_n be the latency of output n. In an original wrapper design, all the inputs in the same iteration are synchronised:

$$\forall m.\, m \in M \Rightarrow \alpha_m = 0 \tag{4.15}$$

$$\forall m, m', k.\, m \in M \wedge m' \in M \wedge k > 0 \Rightarrow t_{m,k} = t_{m',k} \tag{4.16}$$

The time constraint for the output n is then:

$$t_{n,k} \geq t_{m,k} + L_n \tag{4.17}$$

Substituting Equation 4.17 into Equation 4.14:

$$t_{m,k_1} - t_{m,k_2} \geq L_n \tag{4.18}$$

The upper bound of II at m only depends on the latency of the static island and the dependence distance in iterations.

Throughput of Our Proposed Wrapper Design.

We propose a new wrapper that does not require all the input to be valid before the computation. The component can start earlier with some missing inputs as long as these inputs are

[1]The output value may become valid before the whole computation finishes, where the output offset is the time difference.

not required in the next clock cycle, that is, these inputs have non-zero offsets. If an input is required but not valid, the wrapper stalls the whole component until the input becomes valid.

For the new wrapper with positive offsets, Equation 4.16 no longer holds. The output constraint in Equation 4.17 now becomes:

$$t_{n,k} \geq t_{m,k} + L_n - \alpha_m \tag{4.19}$$

Substituting it into Equation 4.14:

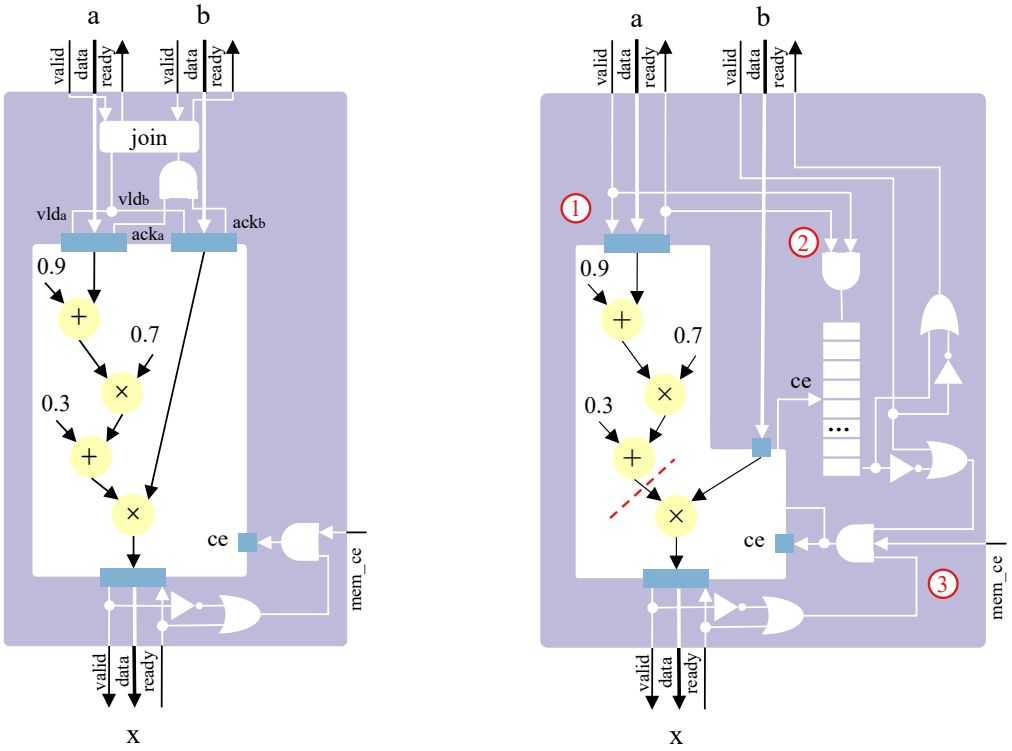
$$t_{m,k_1} - t_{m,k_2} \geq L_n - \alpha_m \tag{4.20}$$

The upper bound of II at m now also depends on its offset and is lower than Equation 4.18 when the offset is non-zero. Our proposed wrapper can avoid significant throughput loss when the offset and latency are both large.

Implementation of Proposed Wrapper.

An example of a static island with two inputs and one output is illustrated in Figure 4.8a. The component takes two inputs `a` and `b` and returns a result `x` as `(((0.9+a)*0.7)+0.3)*b`. When the static island starts to compute, only `a` is required to compute `((0.9+a)*0.7)+0.3` before multiplying with `b`. Assuming each adder has a latency of 4 cycles and each multiplier has a latency of 5 cycles, `a` has an offset of 0 and `b` has an offset of 13.

Figure 4.8a is the original wrapper and Figure 4.8b is our proposed wrapper. Our proposed wrapper is implemented mainly in three parts. First, the interface of an input with zero offsets, such as `a`, is implemented by the same handshake interface as the original wrapper. It checks the valid signal in a constant time interval specified by the II. The component takes *bubble* if the valid is not valid yet, where the data inside the component is *still* being processed [1]. Second, the interface of an input with a positive offset, such as `b`, is controlled by an additional shift register. The shift register is synchronised with the pipeline state of the component. The



(a) Original wrapper

(b) Proposed wrapper

Figure 4.8: An example of a wrapper with offset constraints. `a` is consumed immediately when the component starts to compute (offset = 0), and `b` is only required 13 cycles after `a` is consumed (offset = 13). The original wrapper uses a *join* to synchronise all the inputs. A shift register is implemented in (b) for `b` to count the offset and control its handshake interface. The red dashed line illustrates the state where `b` is required by the component. The implementation of the proposed wrapper contains three parts: 1) Interface for ports with zero offsets, the same as (a); 2) Interface for ports with positive offsets; 3) Interface for backpressure. ce represents the clock enable signal.

time when `b` is required is indicated by a certain bit of the shift register being set, *i.e.* the 13th bit for this example. When the bit is set but `b` is not valid, the whole component is stalled waiting for `b`, where the data inside the component is all stalled.

Finally, the clock-enable signal of the component is used to stall the component. Similar to the original wrapper, it is controlled by the back pressure and memory arbiter. The major difference from the original wrapper is that the absence of `b` can also stall the component.

II Constraints of Proposed Wrapper

The component now can be stalled by both the inputs and the outputs (back pressure). An inappropriate II for this wrapper could cause deadlock. Here we show how to formalise the deadlock problem. Since the inputs may not be synchronised, the condition that always holds for the new wrapper design is:

$$\forall m, m', k.\, m \in M \wedge m' \in M \wedge \alpha_m \geq \alpha_{m'} \wedge k > 0 \Rightarrow t_{m',k} - \alpha_{m'} \leq t_{m,k} - \alpha_m \tag{4.21}$$

This means that an input with a larger offset α_m is always consumed later than another input with a smaller offset $\alpha_{m'}$ by at least $\alpha_m - \alpha_{m'}$ cycles. Then deadlock happens when an executing output is required by an input:

$$\exists m, m', n, k_1, k_2.\, (m, n, k_1, k_2) \in D \wedge m' \in M \Rightarrow t_{m',k_1} + \alpha_m - \alpha_{m'} < t_{n,k_2} \tag{4.22}$$

For instance, when the input m' in iteration k_1 has propagated to the point where the input m is required. The output n in the iteration k_2 is still in the component, and the input m is not valid because of Equation 4.14. The component is forever stalled, waiting for the output that is stalled.

This never happens in the original wrapper because Equation 4.22 never holds under Equation 4.15. In order to avoid the deadlock in the new wrapper, an additional constraint is

Table 4.4: Comparison of total cycles between the designs using the original wrapper and our new wrapper. When there is a dependence between the input and output of a static island, the throughput of hardware using the original wrapper is significantly worse, otherwise only the latency is affected.

Benchmarks	Original wrapper	Our wrapper
vecNormTrans	17439	6703
doitgenTriple	329016	263435
levmarq	54315	54296

required to avoid Equation 4.22:

$$\forall m, m', n, k_1, k_2.\, (m, n, k_1, k_2) \in D \wedge m' \in M \Rightarrow t_{m',k_1} + \alpha_m - \alpha_{m'} \geq t_{n,k_2} \tag{4.23}$$

This always holds when $\alpha_m \leq \alpha_{m'}$. For $\alpha_m > \alpha_{m'}$, substituting Equation 4.19 into the equation above:

$$t_{m',k_1} + \alpha_m - \alpha_{m'} \geq t_{m',k_2} + L_n - \alpha_{m'} \tag{4.24}$$

$$t_{m',k_1} \geq t_{m',k_2} + L_n - \alpha_m \tag{4.25}$$

Assume that a static island is synthesised and pipelined with a constant II of P. Since the input m' is not synchronised with the input m, the actual II of m' is still restricted by P:

$$t_{m',k_1} \geq t_{m',k_2} + P(k_1 - k_2) \tag{4.26}$$

The constraint then becomes:

$$P(k_1 - k_2) \geq +L_n - \alpha_m \tag{4.27}$$

The following must hold for a wrapper design without deadlock:

$$\forall m, n, k_1, k_2.\, (m, n, k_1, k_2) \in D \Rightarrow \{m' \in M | \alpha_m > \alpha_{m'}\} = \emptyset \vee P \geq \frac{L_n - \alpha_m}{k_1 - k_2} \tag{4.28}$$

In summary, the original wrapper assumes no offsets and may have suboptimal performance.

We propose a new wrapper design that has an improved throughput upper bound with II constraints. Our tool checks if the condition holds. If it does not hold and D is data-dependent, the static island is split into multiple static islands with zero offsets. For instance, the static island in Figure 4.8b can be split into two island annotated by the red dotted line. If it does not hold and D is known, the tool relaxes the II of the component until Equation 4.28 holds to avoid deadlock. Even P is restricted, the lower bound of P is still no greater than the optimal II, so the optimal throughput is still reachable. Table 4.4 evaluates the performance improvement by replacing original wrappers with our wrappers. The area change of a wrapper is negligible compared to the total area of the designs.

Tool Flow

Figure 4.2 illustrates the complete tool flow with our work integrated. The input code is analysed by our static island analyser in two steps to determine static islands in loops and instructions. These static islands are then synthesised by Vivado HLS through its LLVM front end [14]. The offset constraints are extracted from the scheduling reports of static islands. The dynamically scheduled code region is transformed into a dot graph buffered with offset constraints, and then translated into a hardware netlist of dynamically scheduled components. Finally, the wrappers with offset optimisation are generated as part of the design.

4.1.5 Experiments

We evaluate our work on a set of benchmarks as introduced in Section 2.5, comparing with the corresponding statically scheduled and dynamically scheduled designs in total circuit area and wall clock time. The IIs of static islands are manually chosen. We obtain the total clock cycles from Vivado XSIM simulator and the area results from the Post Synthesis report in Vivado. The FPGA family we used for result measurements is xc7z020clg484 and the version of the Xilinx software is 2020.2.

Results

Compared with the manual designs by experts over the benchmarks in the original DASS work [1], most of the designs generated by our tool have comparable performance and area as shown in Figure 4.9. Compared with the fastest designs discovered via exhaustive search over the selected ten benchmarks, our approach generates designs that are within 2% of the performance as shown in Fig. 4.10. Details are shown in Table 4.5. The first part of the table shows the detailed results of the six benchmarks where dynamic scheduling should significantly improve the throughput. First, as shown in Section 4.1.2, `vecNormTrans` has high throughput because the input-dependent computation remains in the dynamically scheduled part, and our tool enables resource sharing and LSQ removal. `doitgenTriple` has two loops in a sequence that have memory dependence scheduled by a LSQ. Since static islands cannot share LSQs, these loops remain dynamically scheduled. The loop bodies of two loops can still be synthesised as static islands, resulting in significantly reduced DSPs. A big difference between the DASS hardware and dynamically scheduled hardware is due to the difference in the latencies of floating-point adders in the two tools, which affects the throughput. This also occurs in `levmarq`.

`correlation` has complex loop nests and conditional computation on the standard deviation to avoid zero-divide. Dynamatic pipelines the top-level loop, while Vivado HLS only pipelines the innermost loops. This is more significant for large benchmarks `levmarq` and `gramSchmidt`. The DASS hardware and dynamically scheduled hardware avoid that and achieve higher throughput. There is a significant performance improvement in `gramSchmidt` when switching dynamic scheduling to DASS, because Vivado HLS uses advanced `facc` ops for floating-point accumulation which has a latency of 1 cycle, while Dynamatic uses normal `fadd` ops which has a latency of 4 cycles.

`getTanh` is a special case, where DASS hardware has close performance as statically scheduled hardware but smaller area. In static scheduling, the scheduler assumes all the elements in the input vector are in the linear region, resulting in an II of 1 for the high-precision computation. In DASS, the code region that performs high-precision computation is synthesised as a static

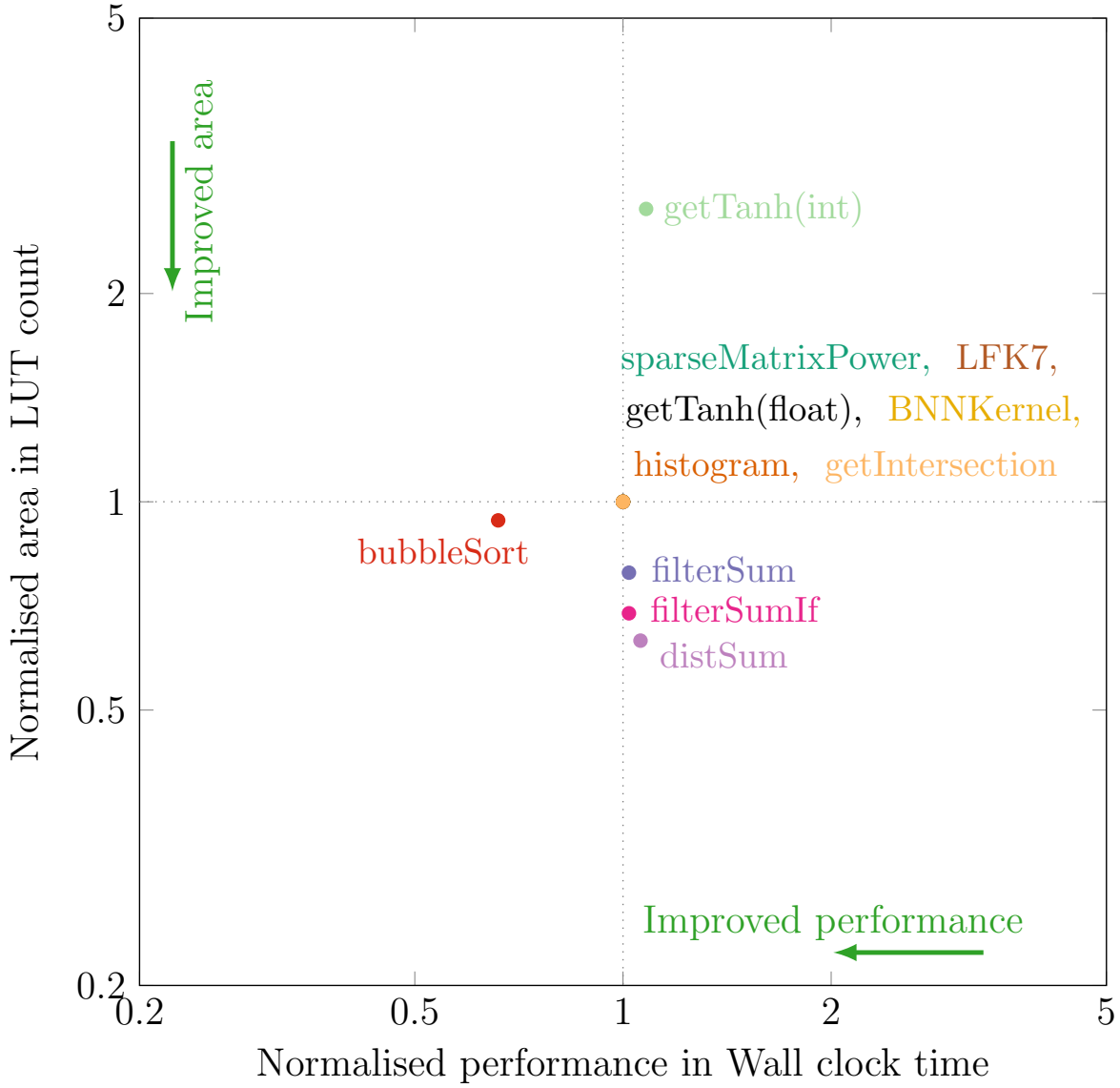


Figure 4.9: Comparison with the manual DASS designs in [1]. The manual DASS designs are at (1, 1) and the points represent the designs by our approach. For most benchmarks, our tool achieves the same hardware as the hardware manually designed by experts. For certain benchmarks, our tool even finds better static islands than those in the manual designs, resulting in smaller area or better performance. The small difference in performance is mainly caused by different maximum frequencies of the designs. We expect higher performance in `getTanh(int)` but the result is restricted by the pipelining capabilities of Dynamatic for complex loops.

Table 4.5: Evaluation of automatically determining static islands on a set of benchmarks. The first part of table evaluates the benchmarks that are amenable to our approach; and the second part of the table evaluates the benchmarks that do not have data-dependent operations. SS = the designs directly synthesised by Vitis HLS. DS = the designs directly synthesised by Dynamatic. DASS = the designs synthesised by DASS using our proposed approach. ADP = area-delay product.

Benchmarks	**IIs**	**LUTs**			**DSPs**			**Cycles**			**Fmax (MHz)**			**Wall clock time (s)**			**Norm. ADP**		
		SS	DS	DASS	SS	DS	DASS	SS	DS	DASS	SS	DS	DASS	SS	DS	DASS	SS	DS	DASS
vecNormTrans	9, 2	1.61k	21.3k	2.86k	5	20	7	12.3k	6.08k	6.70k	121	89.7	106	102μ	63.1μ	67.8μ	1	2.7	0.87
doitgenTriple	5, 5	768	21.8k	20.4k	10	20	10	399k	198k	263k	121	87.6	87.6	3.3m	2.26m	3.01m	1	1.4	0.91
correlation	4, 4, 10, 1, 4	2.13k	14.3k	6.72k	5	36	22	79.9k	66.6k	70.0k	121	80.5	103	661μ	827μ	680μ	1	9	4.5
levmarq	59, 6	2.42k	22.4k	6.74k	15	153	22	204k	51.2k	54.3k	121	120	120	1.69m	427μ	452μ	1	2.6	0.39
gramSchmidt	1, 34, 1, 1, 1, 15, 2	3.27k	15.1k	15.9k	5	49	31	375k	99.0k	61.0k	121	117	117	3.11m	844μ	519μ	1	2.7	1
getTanh	4	892	3.79k	1.32k	16	16	5	1.03k	1.04k	1.03k	121	120	120	8.5μ	8.7μ	8.6μ	1	1	0.31
Norm. median		**1**×	**7.99**×	**2.97**×	**1**×	**5.6**×	**1.43**×	**1**×	**0.50**×	**0.60**×	**1**×	**0.86**×	**0.93**×	**1**×	**0.68**×	**0.76**×	**1**×	**2.6**×	**0.89**×
covariance	4, 6, 4	2.33k	7.91k	4.24k	5	9	9	92.3k	72.9k	84.0k	121	86.1	99.8	764μ	847μ	841μ	1	2	2
syr2k	4	829	4.04k	3.52k	5	19	8	81.4k	66.5k	78.7k	121	98.1	118	673μ	678μ	665μ	1	3.8	1.6
gemver	6, 4, 4	1.44k	8.32k	3.24k	10	28	17	599k	611k	532k	115	106	106	5.23m	5.76m	5.01m	1	3.1	1.6
gesummv	1, 10	2.11k	3.37k	2.75k	8	18	14	71.2k	262k	68.8k	121	113	102	589μ	2.31m	673μ	1	8.8	2
Norm. median		**1**×	**4.14**×	**2.03**×	**1**×	**2.53**×	**1.73**×	**1**×	**0.92**×	**0.94**×	**1**×	**0.87**×	**0.89**×	**1**×	**1.10**×	**1.04**×	**1**×	**3.5**×	**1.8**×

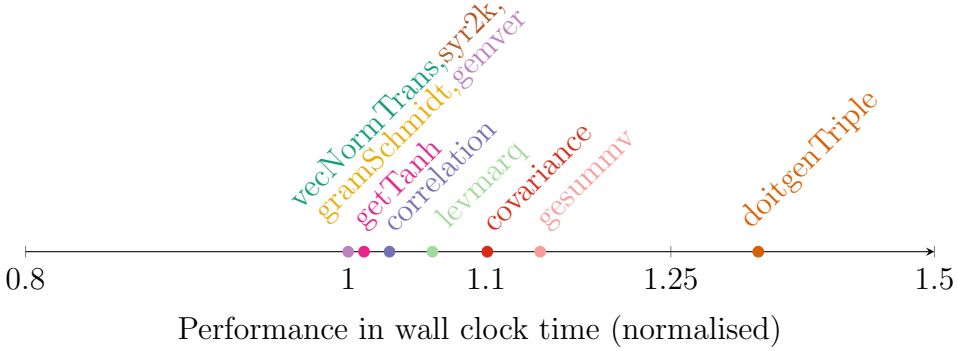


Figure 4.10: Comparison with the designs with the best performance. The best performant design is normalised at 1, and the points represent the relative performance of our designs for evaluated benchmarks.

island. Knowing the probability of the input that is in the saturation region by profiling, the II of the static island can be relaxed. Overall, the average area-delay product of our designs is better than both statically scheduled and dynamically scheduled designs.

The second part of Table 4.5 shows area advantages over naive dynamic scheduling which is separated from those benchmarks where dynamic scheduling provides an advantage. They are more classic benchmarks where static scheduling would be an obvious approach. Overall, the latencies of designs by the three approaches is close but the areas are significantly different. The reduced cycles from dynamic scheduling to DASS is caused by the use of `facc` ops. Our approach is able to reduce the overhead of using dynamic scheduling. However, DASS cannot share resources between two static islands as explained in Section 4.1.4. This results in larger area-delay products compared to statically scheduled designs.

4.1.6 Summary

Existing HLS tools require users to manually specify their scheduling constraints to achieve high-performance and area-efficient hardware designs. In this work, we present a rule-based technique to automatically select the scheduling strategies for the input programs. Our tool

also optimises the interface between code regions using different scheduling strategies to achieve high performance.

We show how to use our approach to automatically determine the static islands in dynamically scheduled hardware. Across a range of benchmarks that are amenable to our approach, our approach achieves 0.13-1.05× the area combined with 0.75-1.61× through automatic identification and synthesis of static islands from fully dynamically scheduled circuits. The performance of the resulting hardware is close to optimum. Our future work will explore the fundamental limits of this approach, both theoretically and practically.

4.2 Balancing Static Islands

In this section, we tackle challenge 3 in Section 3.1.5: How should the scheduling constraints of static islands be optimised using the information from their dynamically scheduled surroundings? The performance of dynamically scheduled hardware can be used to balance the throughput with static islands. However, statically analysing the performance of dynamically scheduled hardware is challenging.

In Section 4.2.1, we explain the motivation and challenges of optimising static islands inside dynamically scheduled hardware. In Section 4.2.2, we work through a simple motivating example, illustrating the challenges in choosing the optimal II for static islands within dynamically scheduled hardware. In Section 4.2.3, we provide background on performance modelling in HLS and Petri nets. In Section 4.2.4, we present our Petri net model for analysing the behaviour of dynamically scheduled hardware. In Section 4.2.6, we evaluate the effectiveness of our approach on a set of benchmarks.

4.2.1 Introduction

In Chapter 3, we proposed and implemented DASS as an HLS tool for FPGAs [1, 130]. The key idea is to take a dynamically scheduled circuit as the starting point, and then to identify

regions of the circuit that can be statically scheduled. These regions, which we call *static islands* [131], often consist of components with fixed or almost-fixed latency, and so benefit little from dynamic scheduling. Each static island is synthesized independently, and a wrapper is placed around it so that it can interface with its dynamically scheduled surroundings. In the case that the whole circuit is one static island, the DASS approach degenerates to static scheduling, and in the case that every static island contains just a single component, the DASS approach degenerates to dynamic scheduling.

However, work on DASS to date has left open the question of how to determine the timing parameters of the static islands, such as their initiation intervals (IIs). An II of a static island is the difference in clock cycles between the start times of its two consecutive iterations. A large II leads to a small area but sub-optimal performance, and a small II leads to high performance but a large area. This section tackles the challenge of automatically determining the IIs for static islands at compile time.

Static analysis for dynamically scheduled hardware behaviour is challenging, because in most cases a program could exhibit many possible state traces depending on the inputs. Existing simulation-based approaches either require an application-specific search algorithm or construct a large design space, which scales rapidly with the computation complexity. This section proposes a probabilistic approach using Petri Nets (PN). The key advantage of using a probabilistic model for dynamic scheduling is that it allows the capture of a large number of possible executions within a very compact representation that can be efficiently explored by existing tools [132, 133, 134]. By modelling this program using the probabilistic graphical representation, we are able to implicitly capture a probability distribution over these traces.

We therefore have two problems: 1) how to adequately model arbitrary program behaviour in this way, and 2) how to use such an efficient description to optimise the resulting hardware. In this section, we explain how we tackle these problems. Our main contributions are as follows:

- We introduce a generic technique to formally describe the scheduling behaviour of HLS-generated hardware in the presence of uncertainty caused by input dependence.

```
1  int A[M], B[N];
2  int ss_func (int x) {
3    return (((((((x+112)*x+23)*x+36)*x+82)*x+127)
4    *x+2)*x+20)*x+100;
5  }
6  int g(int i) {
7    return cond(B[i]) ? i+d:i;
8  }
9  void vecTrans() {
10   for (int i = 0; i < N; i++)
11     A[g(i)] = ss_func(A[i]);
12 }
```

Figure 4.11: Motivating example for inferring the optimal II of a static island. The dependence between `A[i]` and `A[g(i)]` is irregular as it depends on the data in array `B`. This makes it challenging to determine the optimal II for `ss_func`.

- We formalise dynamic scheduling into a Petri net model and explain how static scheduling is integrated into the model.
- We propose an efficient probabilistic analysis to estimate the performance of hardware that has unpredictable behaviours, like input-dependent computations and irregular memory accesses.
- We implement an application of our model to automatically choose the IIs of static islands in dynamically scheduled hardware. Over a set of benchmarks, this only loses up to 6% in area-delay product compared to an exhaustive search of all possible II solutions.

4.2.2 Motivating Example

In this section, we use a motivating example to show the challenge in optimising the II of the static island within dynamically scheduled hardware. Figure 4.11 shows a code example to be scheduled using DASS [1]. In the code, a loop loads an array element `A[i]` and then writes `ss_func(x)` into `A[g(i)]`, where `g` is an external function depending on the loop iterator `i` and an array `B`. Since the store address `g(i)` is unpredictable at compile time, the loop can often be dynamically scheduled to achieve higher throughput than would be attainable statically. However, the function `ss_func` is a polynomial expression, which has predictable timing behaviour. Therefore, function `ss_func` can be synthesized as a static island and enable hardware optimisations inside the island such as resource sharing.

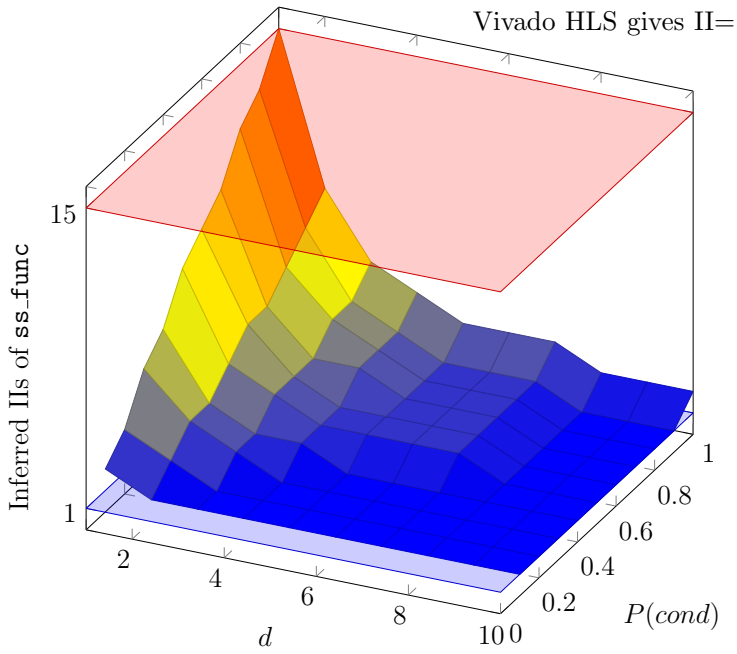


Figure 4.12: The optimal II of function `ss_func` depends on both the probability of `cond` being true and the dependence distance of `d`. The smaller probability when the memory dependence exists or the longer dependence distance the code has, the larger optimal II `ss_func` can have.

A reasonable design objective is to achieve minimum area, subject to the static island `ss_func` not being the performance bottleneck. A larger II may cause performance loss, while a smaller II may cause area overhead. A question arises: How do we estimate the throughput required for a static island like `ss_func` in order for it not to be the bottleneck, and hence determine its II?

Support from existing HLS tools is limited. For instance, the static scheduler in Vivado HLS can only approximate the unpredictable behaviour to the worst case and suggests that the loop should be executed sequentially, corresponding to an II of 15. Meanwhile, Dynamatic has throughput analysis for buffering [110], but the analyser approximates the control flow decisions and ignores memory dependences in the code, resulting in potentially suboptimal II. For example, the throughput analysis in Dynamatic returns a value corresponding to a minimum II of 1 for the example code but this may be unduly optimistic in the presence of memory-carried dependencies.

The above approaches both give a constant II *regardless of the input data*. In reality, the

optimal II of `ss_func` can vary in terms of two constraints: 1) how often `load A[i]` or `store A[g(i)]` in an iteration depends on other memory accesses in previous iterations and 2) if a memory dependence exists, what the dependence distance is in terms of iterations. By varying these two constraints, Figure 4.12 illustrates the distribution of the optimal II over different cases. However, exploring all the design spaces for every hardware design is time-consuming. This section presents a more efficient and accurate solution for finding the optimal II using probabilistic analysis.

Why Petri nets?

Petri nets are a widely used formalism for the modelling and analysis of concurrent processes. When an appropriate time interpretation is considered, they are used for probabilistic analysis, with well-studied techniques in the last few decades [135, 136]. Analysis techniques for Petri nets have been well studied in past decades [136]. By translating our problem into the formal framework of Petri nets, we can rely on existing tools like PRISM [132] and SimHPN [137] or applying known efficient techniques to analyse the resulting Petri nets.

For instance, when the probability that `cond` evaluates to true in Figure 4.12, $P(cond) = 0.4$, $d = 5$, our approach obtains the following results considering the optimal design as the baseline:

Comparison	**Area**	**Performance**
Optimal design	1.00×	1.00×
Analyser in Vivado HLS	0.33×	0.26×
Analyser in Dynamatic	2.33×	1.00×
Our approach	1.00×	1.00×

The search time for exhaustive search to get an optimal II scales with the number of static islands and their II search space. The analysis time for our probabilistic analysis does scale with these constraints and our tool infers all the IIs for static islands in a single analysis. As a result, for the example above, the run time for our process is 0.36× compared to exhaustive search.

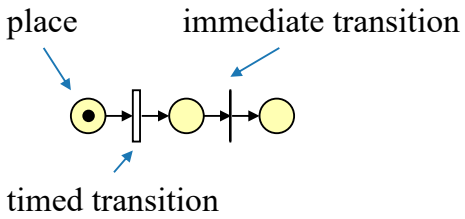


Figure 4.13: An example of a Petri net. The transition on the left has a time delay, and the one on the right has no delay.

Probabilistic analysis can cause inaccuracy in performance modelling, which only affects the performance of the synthesized hardware. However, the correctness of the hardware only depends on the correctness of the synthesis tools themselves. Our analysis suggests an II which will only change the performance or area, while *correctness is always preserved*.

4.2.3 Background

Petri nets are a common mathematical formalism for the modelling and analysis of distributed systems. A Petri net is a directed bipartite graph, consisting of two types of nodes: transitions and places. A transition, usually represented by a bar or a rectangle, represents a process. A place, usually represented by a circle, represents a resource. Places may contain 'tokens', indicated by dots, which represent the state of a resource. The state of a Petri net, known as its 'marking', consists of the overall allocation of tokens to places. Often, places are bounded, meaning that they can only contain at most a certain number of tokens.

In a Petri net, an edge always connects a transition and a place. For each transition, the input places indicate its preconditions, and the output places indicate its postconditions. The transition can only fire when all the preconditions are met, *i.e.* all the input places have tokens and all the output places can take the newly generated tokens without exceeding place bounds.

Most of the Petri net interpretations are "discrete Petri nets", where places hold a positive integer number of tokens, and transitions can be fired in a positive integer amount. However, in a continuous Petri net, a transition net can fire a positive real number of tokens [138], dealing with real tokens in places. Among the different time interpretations which can be

associated with Continuous Petri nets (CPN), we consider time associated with the transitions, with timed transitions under infinite server semantics [134] and immediate transitions, as shown in Figure 4.13. Such a Petri net is called a hybrid Petri net. The closest piece of Petri net interpretation to the one chosen in this work is the hybrid Petri net by David and Alla [138] or Balduzzi and Guia [139]. Both our model and their model include timed transitions and immediate transitions. The main difference is that they restrict immediate transitions to be discrete, while we support all the transitions to be continuous.

Circuit modelling using Petri nets has been investigated for decades [135, 140, 141, 142, 143]. These works model the circuit behaviours using discrete Petri nets, and we use a similar integration philosophy only for performance analysis using continuous Petri nets. We formalise each component from the chosen HLS tools, which automates performance analysis from high-level code. There are also works on performance analysis using Petri nets [144, 145, 146]. These models are for application-specific asynchronous hardware, while we model synchronous hardware for arbitrary code.

4.2.4 Methodology

This section presents our Petri net model for estimating the throughput for dynamically scheduled hardware using which static islands can be optimised. We first explain the specifications of our model. Then we show how to model HLS hardware using the model and analyse the overall performance at compile time. Finally, we show how this analysis can be used to optimise the II of the static islands in the tool flow.

Petri Net Specifications

Here we present the specifications of the Petri nets used in this section. We combine the features from two classical Petri nets, Stochastic Petri nets (SPN) [136] and Timed Continuous Petri nets [134]. In order to adequately model the complex interaction of combinational and sequential behaviour present in modern dynamic HLS tools, we specify two types of transitions,

timed transitions and immediate transitions. A timed transition always fires with a single-cycle delay, and an immediate transition always fires with no delay. Figure 4.13 shows an example of a Petri net containing three places, a timed transition and an immediate transition. The place on the left holds a token that can enable the timed transition in the next clock cycle. At that point, the immediate transition would be enabled and it will immediately fire, so the token will immediately reach the place on the right.

The timed and immediate transitions are simply a way of modelling systems consisting of combinational logic and sequential logic. This enables us to model arbitrary hardware behaviour in clock cycles regardless of the hardware being statically or dynamically scheduled.

The specifications of our Petri net model are extended from the formulation by Murata [147]. Our Petri net is a 7-tuple, $N = (P, T, N, E, E', W, M_0)$ where:

- $P = (p_1, p_2, p_3, ..., p_i)$ denotes a vector of i places,
- $T = (t_1, t_2, ..., t_j)$ denotes a vector of j transitions,
- $N : T \rightarrow \{\texttt{T}, \texttt{I}\}$ denotes the transition type, which could be either timed (`T`) or immediate (`I`),
- $E \in \{0, 1\}^{|P| \times |T|}$ denotes a matrix of edges from places to transitions,
- $E' \in \{0, 1\}^{|P| \times |T|}$ denotes a matrix of edges from transitions to places,
- $W : \{E, E'\} \rightarrow \{0 \leq x \leq 1 | x \in \mathbb{R}\}$ denotes the probability of an edge to execute with respect to the other edges from the same place, and
- $M_0 \in \mathbb{N}^{|P|}$ denotes the initial marking.

In this model, we approximate program behaviour by only considering the presence/absence of data at a particular place – as indicated by a token – rather than its value, approximating data-dependent operations by probabilistic execution. A place with tokens indicates the presence of data held by a component. A transition indicates the computation of a component. The probability function models the probability of triggering an edge when its adjacent transition

is enabled, so the sum of all the probabilities of edges from a place is 1. The initial marking M_0 describes the number of tokens contained in each place at the initialisation of the hardware.

Petri Net Modelling

Here we show how to model both dynamically and statically scheduled hardware components. We also model the back pressure by the handshake signals in the hardware.

Modelling Static Islands

A static island has a predictable hardware behaviour by static scheduling. It has a constant II and a constant latency. From the scheduling report, we extract the parameters of a static island S shown in Equation 4.29, and use them to transform the circuit into a Petri net N.

$$S = (n_{\text{in}}, n_{\text{out}}, II, l) \tag{4.29}$$

n_{in} and n_{out} are the numbers of inputs and outputs of the static island. II and l are the initiation interval and latency.

For instance, the Petri net on the right side of Figure 4.6 models a static island. The yellow places represent the data path in the circuit; the purple places represent the back pressure and the white places represent other control signals to meet the specifications.[2] Back pressure happens when the component cannot accept an input from its preceding component, which causes a pipeline stall.

The model contains n_{in} yellow places for representing the data inputs and n_{in} purple places for representing the corresponding back pressure. A token in one of these yellow places represents the arrival of data at that input. A token in one of these purple places represents that the component can accept data at that input from its preceding component.

Similarly, the model contains n_{out} yellow places for representing the data outputs and n_{out} purple

[2]The colours of places are for annotation only. All the places are treated the same in the tool.

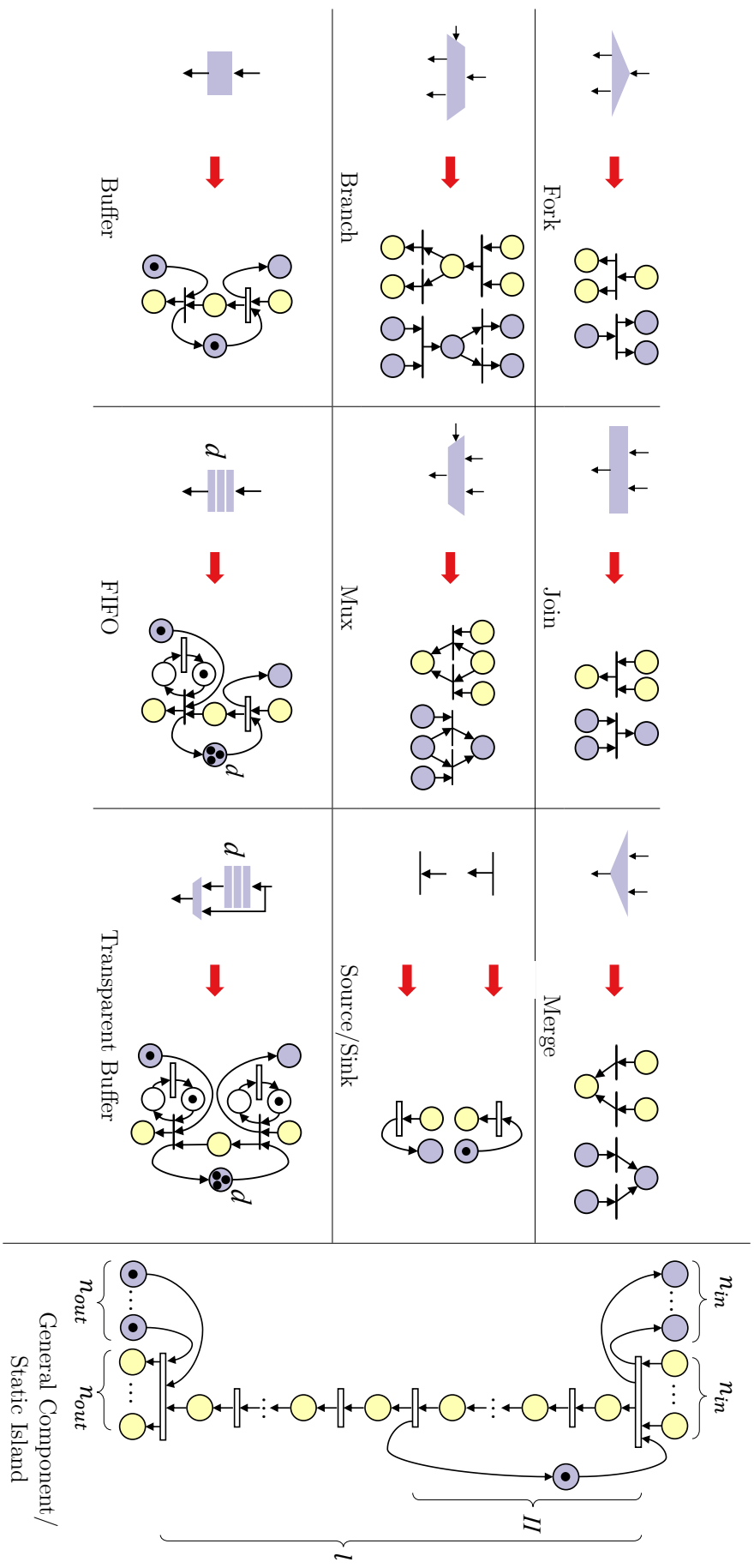


Table 4.6: Component Formalisation of HLS Hardware.

places for representing the corresponding back pressure. A token in one of these yellow places represents the validity of data at that output. A token in one of these purple places represents that the component can propagate data from that output to its succeeding component.

The adjacent connection to the input places ensures that the static island can only start to compute when all the inputs hold valid data. The path between the inputs and outputs consists of l timed transitions corresponding to the latency of the static island in clock cycles.

Finally, a back edge from the IIth timed transition to the input transition constructs an internal loop for modelling an II. The initial token in the back edge ensures that there is always at most one token flowing in the loop, limiting the maximum throughput of the hardware to $1/II$. Such a formulation of S in Equation 4.29 enables modelling the time behaviour of a statically scheduled pipeline using Petri nets.

Modelling Dynamically Scheduled Components

We now model the hardware behaviour of dynamically scheduled hardware using the Petri net above. Dynamically scheduled HLS tools generate a graph consisting of several pre-defined components. These components can be divided into three types: control components, general components and memory components. In this section, we show how to formalise these components using Petri nets.

Control Components

Control components parallelize the computation and determine the control flow of the circuit. Here we utilise Dynamatic [13] components. In Dynamatic, the main control components are shown in Table 3.1:

Fork replicates the data into multiple copies to the consumers.

Join stalls until all the inputs hold valid data.

Merge sends the data from one of the inputs to the output.

Branch sends the data to one of the outputs selected by the condition bit.

Mux selects the data from the input determined by the select bit to the output.

Source/Sink constantly sends/accepts data.

The Petri net models of these components are shown in Figure 4.6. In the figure, the symbols on the left-hand side of the red arrows represent the components in the circuit, and the symbols on the right-hand side of the red arrows represent the corresponding Petri net models. A Petri net of these components consists of two parts, the data path and the control path (back pressure). The data path propagates data from its inputs to its outputs, and the control path propagates back pressure from its outputs to its inputs. For instance, the transition in the *join* model only fires when each yellow place at the input holds a token, corresponding to the presence of valid data, and each purple place at the input holds a token, corresponding to the absence of back pressure. The *source/sink* model contains a timed transition, which can process at most a set of data in each clock cycle.

Buffer Components

There are three types of buffers in dynamically scheduled hardware for improving the throughput of the circuit as shown at the bottom of Figure 4.6. A normal buffer is modelled based on the following specifications:

1. it accepts at most a set of data in each clock cycle,
2. it outputs at most a set of data in each clock cycle,
3. it has a latency of one clock cycle, and
4. it can contain at most one set of data.

The Petri net model is then the same as a static island with $S_{\text{Buff}} = (1, 1, 1, 1)$. The timed transition at the input ensures condition 1 always holds. The purple place in the back edge initially contains a token, restricting at most one token in the component for satisfying condition

4. The immediate transition at the output ensures condition 3. Condition 2 always holds because of 4, where there is at most one token at the input in each clock cycle.

The second buffer type is a FIFO, which has a depth greater than 1. A FIFO is modelled based on the following specifications:

1-3) it has the same conditions as conditions 1-3 for a normal buffer, and

4) it can contain at most d sets of data, where d is the FIFO depth.

Compared to the Petri net of a normal buffer, the purple place (annotated with d) in the back edge has d initialised tokens, allowing multiple sets of data inside the FIFO. This ensures condition 4. To restrict the output throughput for satisfying condition 2, a timed loop is added onto the immediate transition at the output. The timed loop contains a token and a timed transition, which ensures that the immediate transition can fire at most once.

Finally, a transparent buffer acts as a FIFO with a combinational delay, leading to better latency. The specification is as follows.

1-2) it has the same conditions as conditions 1-2 for a normal buffer,

3) it has a latency of zero clock cycle, and

4) it can contain at most d sets of data, where d is the depth.

The latency of zero clock cycles means that the data path from the input to the output must not contain any timed transition. Therefore, the input transition becomes immediate for satisfying condition 3. In order to satisfy condition 1 and 2, timed loops are added on both the input and output immediate transitions. Condition 4 holds because of the same setup as the FIFO.

Generic Components The general components such as arithmetic operators and logical operators in the dynamically scheduled hardware compute the data values using a static control flow. They are modelled the same as static islands on the right of Figure 4.6. Each component is

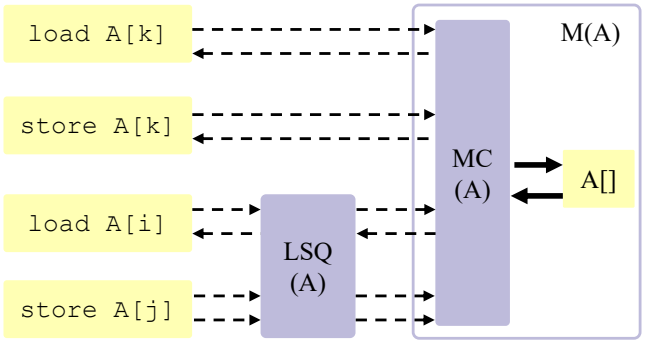


Figure 4.14: A memory controller (MC) is used for balancing the memory bandwidth, and a load-store queue (LSQ) is used for dependence control.

modelled based on Equation 4.29. If a component is combinational, it only has immediate transitions and no back edge.

Memory Components For on-chip memory accesses, there are three main types of memory components in dynamically scheduled hardware: **memory controllers (MC)**, **memory nodes** and **load-store queues (LSQs)**. Figure 4.14 illustrates an example of the memory architecture of dynamically scheduled hardware. The yellow nodes are the memory nodes in the hardware. The purple blocks are the MC and the LSQ. Dynamatic automatically performs aliasing analysis on these nodes to decide whether a node should connect to an LSQ [61]. For this particular example, the values of `k` cannot overlap with the values of `i` and `j`. That is, `load A[k]` and `store A[k]` are independent from `load A[i]` and `store A[j]`. The memory nodes that cannot have conflicts with others like `load A[k]` directly connect to MC, while the other nodes like `load A[i]` are scheduled by the LSQ before reaching the MC.

The Petri net model of Figure 4.14 is shown in Figure 4.15. We first model each memory component in Petri nets, and then compose the Petri nets to model the whole memory architecture.

An **MC** serialises the memory requests from memory nodes. In Dynamatic, each load or store statement in the program is synthesized as a memory node in hardware. Each array is synthesized into a two-port BRAM, a port only connecting load nodes and a port only connecting store nodes. Each BRAM block allows at most one load and one store in every clock cycle. The MC acts as a load arbiter and a store arbiter. Each arbiter has a latency of one clock cycle, indicated by the timed transition in the MC.

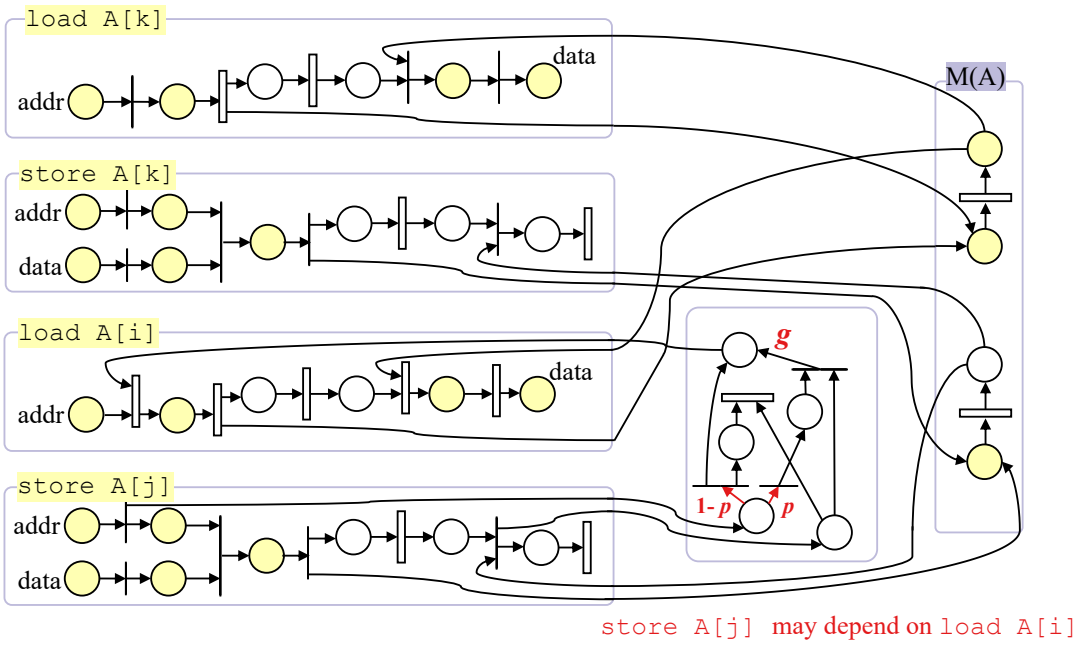


Figure 4.15: All the memory nodes are directly connected to MC. The LSQ is modelled as probabilistic dependence among memory nodes. The latency of a `load` without an LSQ is 2, and the latency of a `load` with an LSQ is 5.

A **memory node** sends requests to an MC and gets the signals back. The Petri net models of `load`s and `store`s are also pre-defined models like those in Figure 4.6. For instance, a token enters `load A[k]`, passes through a transition, and fires two tokens at the output. One goes to the MC, and one goes through the internal path inside the load component. Once the request is granted by the MC, the output token from MC is sent back to `load A[k]` among all load nodes based on the presence of the token in `load A[k]`

There is always at most one token held among all load nodes since the MC serialises the requests. The loads connected to the MC and LSQs have the same Petri net model but different latencies represented as the number of timed transitions in Figure 4.15. For stores, the model is similar to the loads but has two input places representing address and data, respectively. The returned token from the MC only represents an acknowledgement signal.

An **LSQ** schedules memory accesses in terms of dependence. For every two memory accesses connected to an LSQ, the LSQ checks at run time whether there is a dependence. For example, in Figure 4.15, `load A[i]` may depend on `store A[j]`. The LSQ for this dependence is modelled as LSQ(A) which processes the states of `store A[j]` and returns a control signal

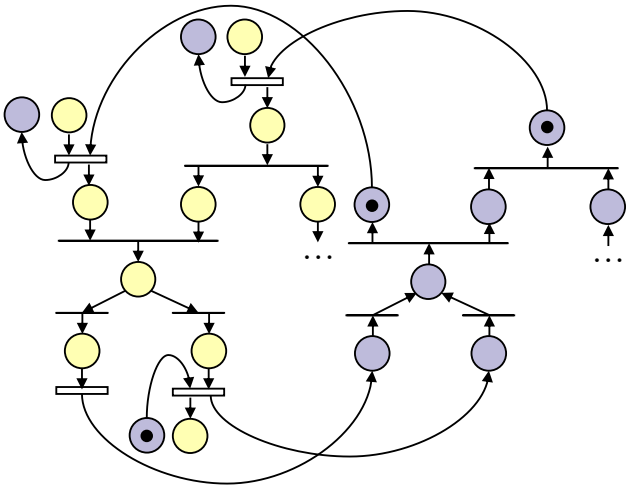


Figure 4.16: The hardware is modelled by directly stitching up the component models. Yellow places are overlapped with other yellow places, and purple places are overlapped with other purple places.

to enable `load A[i]` to compute. In Figure 4.15, whenever `store A[j]` starts to compute, the LSQ processes a token from the address place. The token can take one of the two paths that both reach the place g that determines whether `load A[i]` can compute. If there is no dependence, the token takes the left path and immediately arrives at g, enabling the `load A[i]`. However, if the dependence exists, the token takes the right path. It gets stalled by a *join* until the completion of `store A[j]`, indicating the existence of a dependence. The dependence distance is modelled by the number of initialised tokens k in place g. These tokens allow the `load A[i]` to run k iterations ahead if a dependence occurs. The probability p of `load A[i]` depending on `store A[j]` is modelled as the probability function of the edge.

For the case where `store A[j]` also depends on `load A[i]`, another LSQ block in Figure 4.15 is added to the Petri net but pointing from `load A[i]` to `store A[j]`. We analyse every pair of memory nodes that connect to the same LSQ to capture all possible memory dependences.

Stitching Together

DASS automatically maps the input program into a dataflow graph of components, where all the components in the graph are modelled as above. The hardware can be modelled by translating each component into Petri nets and connecting them by overlapping the input/output places of these components. Figure 4.16 shows an example of a hardware model by stitching up the components. The yellow places are overlapped with yellow places for modelling the data path,

and the purple places are overlapped with purple places for modelling the back pressure.

The back pressure also ensures the behaviour of the synchronous dataflow circuit, where combinational components cannot hold data. In a Petri net, the corresponding constraint is that for any path between any two timed transitions, there are always at most $1 + \sum t + \sum d$ tokens. $\sum t$ is the number of timed transitions in the path, and $\sum d$ is the total depth of the FIFOs in the path.

Petri Net Analysis

We now show how to estimate the overall throughput of the dynamically scheduled hardware by analysing the steady state of the Petri net obtained from Sec. 4.2.4.

Steady State Analysis of Petri nets

We analyse the steady state of a Petri net to estimate the overall hardware performance. The steady state of a Petri net represents its most commonly executing states during the computation, which indicates the overall hardware throughput in our model.

For a discrete Petri net, the analyser first constructs a reachability graph of the given Petri net. A reachability graph contains a finite number of vertices and edges. Each vertex represents a reachable state of the Petri net, and each edge represents a transition between states. The analyser translates the reachability graph into a Markov chain. The Markov chain is further translated into a linear programming problem by the probabilistic analyser such as PRISM [132] for determining the steady state. Discrete Petri nets suffer from the well-known *state explosion problem*, as its reachability space grows exponentially with respect to their initial marking [138].

The steady-state analysis of continuous Petri nets significantly increases the scalability by skipping the process of constructing the reachability graph. Analysing hybrid Petri nets that support timed transitions and immediate transitions has been studied [138, 139], but only as hybrid Petri nets, while we consider them as continuous Petri nets, and there is no existing

tool automating the process. The closest work providing an implementation is the SimHPN analyser by Júlvez *et al.* [137] but it only supports timed transitions under infinite servers semantics. Such an approach does not work for the analysis of immediate transitions. We implement an efficient tool that automatically determines the steady state of continuous Petri nets that contain immediate and timed transitions.

We use the specifications of continuous Petri nets [134] to model the computation of the hardware design. However, the initial marking must be discrete to model the real hardware computation. We aim to improve the scalability of the steady-state analysis, and minimise the inaccuracy caused by this approximation.

The use of continuous Petri nets significantly accelerates the steady-state analysis of our Petri net model. It does not require constructing the reachability graph for steady-state analysis. The discrepancies caused by approximating token flows from integer numbers to real numbers are negligible compared to the ones caused by approximating the values of data to the presence of data and probabilities. Our results in the later section show that using continuous Petri nets can generate the same result but at a significantly faster speed compared to using discrete Petri nets.

Relaxing discrete PN to continuous PN obtains more tractable analysis techniques, at the price of losing some fidelity such as losing some properties. In particular, a deadlock-free discrete PN could deadlock as continuous, as a non-reachable deadlock marking holding the state equation could become reachable [148]. Those potential *spurious* markings are the vertices of the polytope of reachable markings and in most cases it would not be computed as the steady state. Indeed, our implementation does not observe any deadlock during all the experiments reported in Section 4.2.6. In the rare event, a deadlock is detected, our tool flow will switch to the discrete Petri net for steady-state analysis.

Steady State Formulation

We extend the terms defined by Silva *et al.* [134] to model the computation of our Continuous Petri net model with timed and immediate transitions.

- $C = E' - E$ denotes the edges of the Petri net in a matrix,
- $\sigma \in \mathbb{R}_{\geq 0}^{|T|}$ denotes the firing count of the transitions,
- $f \in \mathbb{R}_{\geq 0}^{|T|}$ denotes the flows of the transitions,
- $m \in \mathbb{R}_{\geq 0}^{|P|}$ denotes a run-time marking, and
- $\alpha \in \mathbb{R}_{\geq 0}^{|P|}$ denotes the marking rate in the places.

The flow of a transition f_t is the derivative of its firing count σ_t, where $f = \dot{\sigma}$ [134]. The flow indicates the firing rate of the transition. We propose a new term α to represent the rate that a token passes through a place, which correlates to the firing rates of the input and output transitions of the place.

We now introduce our steady-state formulation using the given constraints above. First, the flows of transition must not change the marking at the steady state.

$$C \cdot f = 0 \tag{4.30}$$

Second, the marking at run time must be reachable from the initial marking after a set of firing iterations of transitions.

$$m = M_0 + C\sigma \tag{4.31}$$

Third, the flow of a timed transition is affected by the presence of data in its input places. It is restricted by the minimum number of tokens among the input places of the transition. Additionally, it is also affected by the synchronous hardware behaviour where any synchronous

operation must process at most a set of data in each clock cycle. The flow of any timed transition must not be greater than 1.

$$\forall t, p.\, N(t) = \mathtt{T} \wedge E_{p,t} > 0 \Rightarrow f_t \leq \min\{1, m_p\} \tag{4.32}$$

Constraint 4.32 restricts the flows of timed transitions. However, the flows of immediate transitions can be infinite. The flow of an immediate transition is not restricted by the presence of data in its input places since a token can go through multiple immediate transitions in a clock cycle. Even though the flow of an immediate transition can be infinite based on our specifications in Sec. 4.2.4, the flow could be restricted by the flow of the neighbour transitions of the immediate transition.

In order to restrict the flow of immediate transitions, we use the marking rates α to pass the flows of these neighbour transitions through places. In a Petri net, we define the marking rate of a place as the sum of the flow of its input transitions. We use the following constraint to model α.

$$\forall t, p.\, \alpha_p = \sum_{E'_{p,t}>0} f_t \tag{4.33}$$

We then use the marking rate of a place to restrict the flow of its output transitions, which could be immediate. The flow of a transition is affected by all the edges from its input places. This includes two constraints, the probability of the edge and the marking rate of a place at the tail of the edge. The probability of the edge represents the portion of the marking rate in the place that could trigger the current transition. The flow of a transition is then restricted by the probability-weighted marking rate of each input place.

$$\forall t, p.\, E_{p,t} > 0 \Rightarrow f_t \leq \alpha_p W(E_{p,t}) \tag{4.34}$$

With the constraints above, our tool automatically searches for the maximum of $\sum f$ and explores the overall throughput of each component at the steady state.

4.2.5 Tool Flow

This section illustrates how our work is integrated into DASS. First, the input C program is lowered into LLVM IR using Clang and gets optimised by the front end of DASS at the software level. Second, the static islands are determined based on user annotations in the code. On the right of the figure, the dynamic scheduled part of the code is then translated to a dataflow graph. Each node in the dataflow graph represents an operation and each edge represents a dependence. Each static island is mapped as a single component in the dataflow graph. Our work starts to optimise the hardware as a plugin:

1. The dataflow graph is buffered for the best throughput, assuming that each static island has a minimum II;

2. The dataflow graph is then translated into a Petri net;

3. Our analyser runs steady state analysis and estimates the average throughput of each static island, leading to an inferred II;

4. These IIs are used as constraints for the static islands for the final hardware generation by Vivado HLS;

5. Also, the IIs are added to the unbuffered dataflow graph for re-buffering, since the existing buffering solution may not work; and

6. The buffering tool re-buffers the dataflow graph for the final hardware generation.

Finally, the synthesis tools generate the corresponding parts of the RTL code and all the output files form as the final hardware design. Vivado HLS generates the code of static islands. Dynamatic translates the dataflow graph into a hardware netlist with pre-defined dynamically scheduled components. The DASS back end generates the corresponding wrapper for communications between static islands and dynamically scheduled hardware.

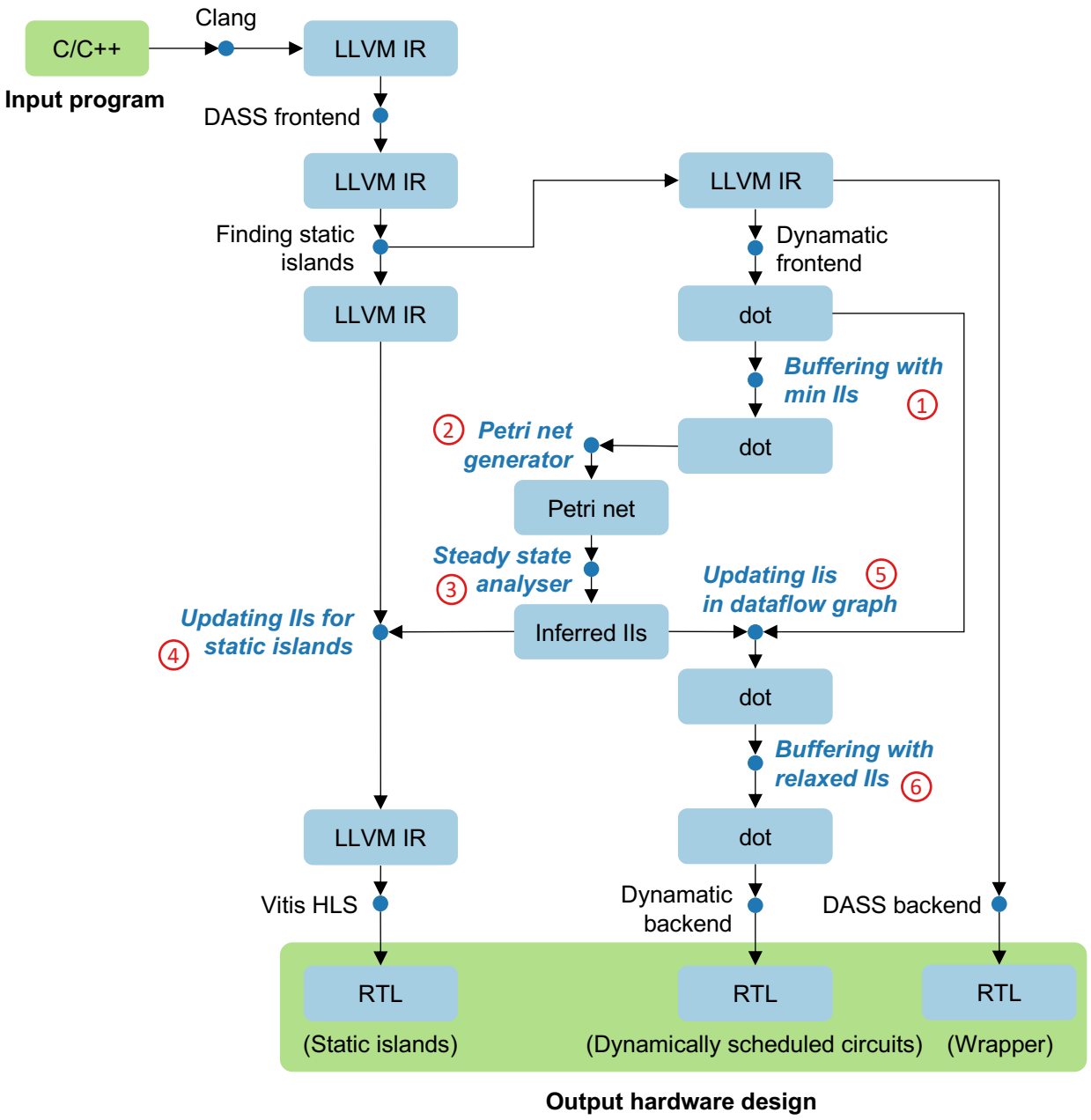


Figure 4.17: Our work of Petri net modelling and analysis integrated into the open-sourced DASS HLS flow. Our contributions are highlighted in blue text.

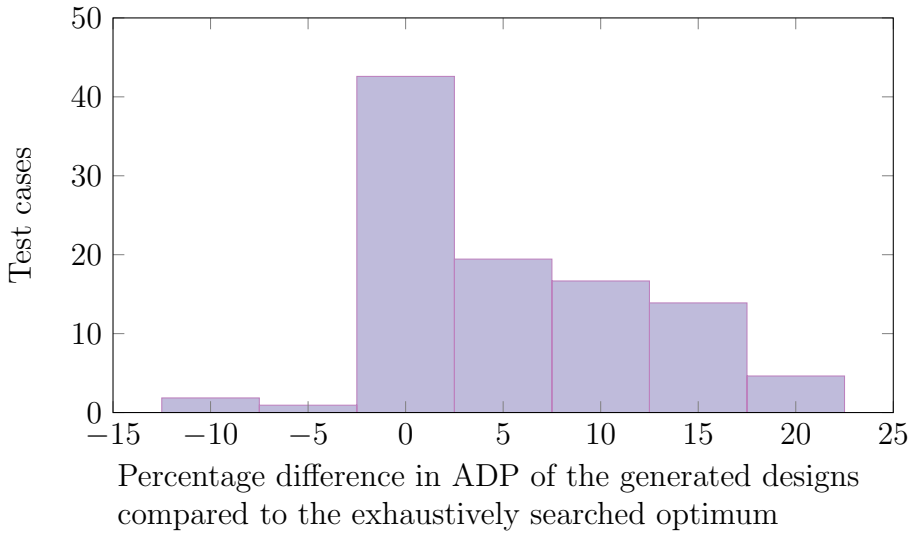


Figure 4.18: Area-delay product (ADP) difference between our work and the optimal designs for `vecTrans` over a set of input distributions. Small differences mean better design quality.

4.2.6 Experiments

We evaluate our approach on the latency and the area of the whole hardware compared to the IIs selected by the static analysis in Vivado HLS, the throughput analysis in Dynamatic and exhaustive search. Over a set of benchmarks, we assess the impact of our analysis on both the circuit area and the performance. We obtain the total clock cycles from ModelSim 10.6d and the area results from the Place & Route report in Vivado. The FPGA family we used for result measurements is xc7z020clg484, and the version of Vivado software is 2020.1.

Results

We first evaluate our approach on the motivating example `vecTrans`. In order to get the most likely optimal II as references, we exhaustively enumerate all the possible IIs for each static island for each input data distribution. The exhaustively searched II is selected as the largest II with a latency of no more than 110% [3] of the minimum latency among all the IIs. Then we compare the searched IIs with the inferred II by our tool for each case.

The distribution of inferred IIs is similar to Figure 4.12 with small variations. Figure 4.18 shows a histogram of the comparison between our inferred IIs and the exhaustively searched IIs in

[3] 110% is selected due to small latency noise caused by the wrapper around the static island [1].

area-delay product (ADT) for `vecTrans`. We test 110 cases by varying the input data, resulting in different sets of probability constraints. A small difference means better design quality. In 86% of the cases, the designs by our approach have less than 10% difference compared to the designs by exhaustive search. The reason for the negative difference is that the latency noise in the static island wrapper affects the selection of the optimal II. There are still 14% cases where the difference in ADP is greater than 10%, because our tool approximates the memory architecture into probabilities instead of the exact sequence.

Table 4.7 shows the results for all the benchmarks selected from Section 2.5. For each benchmark, we use a set of randomly-generated data that is not an extreme case. We compare the area and delay of the designs with the inferred II by our tool to the designs with the IIs suggested by Vivado HLS (base 1), the IIs manually calculated from Dynamatic throughput analysis (base 2) and the optimal IIs by exhaustive search using simulations (search). The II inferred by our tool causes up to 6% overhead in ADP compared to the exhaustively searched II, while the ADP for the II by Vivado HLS causes up to 528% overhead in ADP, and the ADP for the II by the throughput analysis in Dynamatic causes up to 62% overhead in ADP.

There are certain cases where the II from Vivado HLS or Dynamatic has comparable results with the exhaustively searched II. For instance, `evalPos` does not have memory dependence but conditional loop-carried dependence in the code. Such low complexity enables Vivado HLS to suggest a smaller II smaller than the optimal II. Since the code size of the benchmark is small, II of 5 allows the hardware design to share most of the resources. The same for the designs with the IIs by Dynamatic. These cases usually happen for designs where the control flow and memory accesses have low complexity.

The total compilation time of the tool flow is also evaluated in Table 4.8. We compare our approach with exhaustive search and our prior discrete Petri net approach [149]. The time for exhaustive search using simulations depends on both the synthesis and simulation time, and it scales exponentially with the number of static islands and the number of possible IIs, such as `levmarq`. Our continuous Petri net-based approaches significantly reduce the scalability issue by avoiding enumerating the IIs. Additionally, our continuous Petri net-based approach

Table 4.7: Evaluation of automatically inferring IIs for static islands on a set of benchmarks. base 1 = the designs with IIs conservatively chosen by Vivado HLS, base 2 = the designs with IIs manually inferred from Dynamatic throughput analysis, ours = the designs with IIs inferred by our model and search = the designs with IIs by exhaustive search. The LUT count is for the whole design except the LSQs which do change with these approaches [13]. Optimising the LSQs is a separate problem.

Benchmark	**II**				**LUTs**				**DSPs**				**Cycles**				**Fmax (MHz)**			
	base 1	base 2	ours	search	base 1	base 2	ours	search	base 1	base 2	ours	search	base 1	base 2	ours	search	base 1	base 2	ours	search
vecTrans	15	1	3	3	785	759	922	922	3	21	9	9	101k	58.5k	60.1k	60.1k	112	104	79.6	79.6
vecTrans2	15	6	6	7	150	470	470	434	3	6	6	3	40.7k	24.2k	24.2k	24.5k	58.6	60.0	60.0	62.7
vecTrans3	49	1	2	2	1.22k	2.59k	2.68k	2.68k	5	27	15	15	196k	14.4k	14.2k	14.2k	102	100	97.7	97.7
evalPos	5	10	11	14	3.63k	3.68k	3.63k	3.55k	14	14	14	14	2.31k	2.24k	2.24k	2.24k	80.5	77.0	80.4	80.4
levmarq	59,72	1,2	59,6	59,8	2.86k	7.01k	4.141k	4.141k	26	75	31	31	1.34	928k	936k	969k	63.8	54.2	61.2	61.2
chaosNCG	74	1	8	28	3.61k	5.67k	3.78k	3.44k	0	0	0	0	400k	140k	146k	151k	77.4	93.0	92.5	87.9
Normalised geom. mean	-	-	-	-	**0.74**×	**1.21**×	**1.03**×	**1**×	**0.75**×	**1.76**×	**1.17**×	**1**×	**3.71**×	**0.98**×	**0.99**×	**1**×	**1.05**×	**1.03**×	**1**×	**1**×

Table 4.8: Synthesis time comparison between exhaustive search and our approach. The colour bar at the bottom illustrates on average the ratio of synthesis time, simulation time and our analysis time for a single design with an II. sim = simulation-based; dpn = discrete Petri net analysis; cpn = continuous Petri net analysis.

Benchmarks	Synth (s)	Analysis (s)			Total (s)		
		sim	dpn	cpn	sim	dpn	cpn
vecTrans	189	3.96k	1.08k	1	3.96k	1.46k	379
vecTrans2	165	3.46k	10	2	3.46k	340	332
vecTrans3	187	11.8k	1.08k	2	11.8k	1.45k	376
evalPos	212	13.2k	9	8	13.2k	433	432
levmarq	560	2.70M	1.09k	51	2.70M	2.21k	1.17k
chaosNCG	187	23.0k	1.08k	11	23.0k	1.45k	385
Geom. mean (speedup)		1×	14×	2.8k×	1×	8.7×	22×

achieves even more speedup compared to the discrete Petri-net based approach. Overall, we achieve a 22× speedup in the total compilation time for synthesizing a hardware design with optimised IIs.

4.2.7 Summary

In high-level synthesis, static analysis for scheduling is usually carried out based on worst-case assumption or exhaustive search. Efficient modelling dynamic mechanism in static analysis for hardware is challenging. In this work, we present a technique to translate the uncertainty of hardware behaviours, such as data-dependent choices and unpredictable memory accesses, into a probabilistic model. We rely on pre-existing tools based on Petri nets for analysis and optimisation.

We show how to use our model to automatically estimate the optimal II of the static islands in dynamically scheduled surroundings. Across a range of benchmark programs that are amenable to DASS, our approach achieves 10.4-316k× speedup compared to the design by exhaustive search on estimating an optimal II with up to 6% overhead in ADP, while the static analysis in Vivado HLS causes 528% overhead in ADP, and the throughput analysis in Dynamatic causes 62% overhead in ADP. Our future work will explore the fundamental limits of this approach, both theoretically and practically.

Chapter 5

Parallelising Control Flow

Chapter 3 formalised the dependence model for dynamic scheduling and identifies the restriction for BB execution in existing dynamic scheduling implementation:

$$\forall i, b, k.\, i \in I_b \wedge (b,k) \in E_B \Rightarrow t(b,k) \leq t(i,k) \tag{5.1}$$

$$D : \forall e, e'.\, e \in E_B \wedge e' \in E_B \wedge e \prec e' \Rightarrow t(e) < t(e') \tag{5.2}$$

The dependence constraint could be relaxed to the following as explained in Section 3.1.5:

$$D'' : \forall b, b', k, k'.\, (b,k) \in E_B \wedge (b',k') \in E_B \wedge (b,k) \prec (b',k') \wedge d'(b,b') \Rightarrow t(b,k) < t(b',k') \tag{5.3}$$

This chapter reformulates the publication at FCCM 2022 and FPL 2022 for the formulation provided in Chapter 3. First, we extend the FCCM 2022 publication by reformulating the problem in the generic model and show how to achieve out-of-order BB execution for C-slow pipelining. Second, we extend the FPL 2022 publication by similar reformulation and show how to achieve simultaneous BB execution for block-level scheduling.

5.1 Dynamic C-slow Pipelining

This section demonstrates the first application of the relaxed dependence formalisation above for achieving out-of-order execution of independent and consecutive iterations of the same BB by C-slow pipelining in nested loops.

In Section 5.1.1, we explain the motivation and challenges of enabling C-slow pipelining. In Section 5.1.2, we demonstrate a motivating example of C-slow pipelining the innermost loop of a loop nest. In Section 5.1.3, we provide background about C-slow pipelining. In Section 5.1.4, we show how to use Microsoft Boogie to automatically determine the absence of dependence between loop iterations and how to use the analysis for C-slow pipelining in hardware. In Section 5.1.5, we evaluate the effectiveness of our approach on a set of benchmarks.

5.1.1 Introduction

Static pipelining enables efficient resource sharing thanks to its predictable execution timing. However, it also causes conservatism in input-dependent dependence analysis which may result in suboptimal throughput. Dynamic pipelining uses a handshake interface between operations which allows immediate execution when an operation has all its required data. However, this also leads to input-dependent hardware behaviour, making resource sharing challenging.

In dynamic pipelining, each operation is synthesized as a single component. These components are stitched together to form the top-level hardware design. Each data dependence between two operations is presented as a handshake connection between them. The memory dependences, however, are handled either by sequentialising all the memory accesses [2], or by using load-store queues (LSQs) to schedule the memory accesses at run time [60]. The former keeps the conservatism in static analysis, which can lead to poor performance. The latter checks the dependence between every pair of memory operations in the original program order and allows the later memory operation to execute earlier if dependences allow. Using LSQs enables out-of-order memory accesses at run time, achieving better performance.

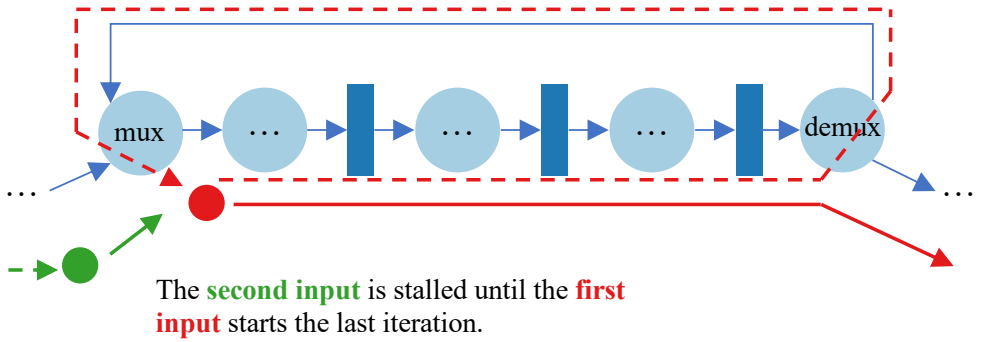


Figure 5.1: An example of a dynamically pipelined loop starting iterations in order. The red token is the first input and the green token is the second input. Assume the second input has no dependence with the first input and the loop computes with an II of 3. The restriction of sequential control flow forces the second input to stall until the control flow of the first input finishes. Our approach enables C-slow pipelining and allows the second input to start right after the first input, leading to a doubled throughput.

The LSQ used in Dynamatic [13], a representative dynamically scheduled HLS tool, reduces the conservatism in static pipelining. However, it still has a restriction that the hardware must preserve the original program order of memory accesses when it checks dependences at run time. Dynamatic chooses to statically schedule the memory accesses in each BB, and dynamically allocates these memory accesses into the LSQs when the corresponding BB starts to execute [60]. That means that there is always *at most one* BB *starting* at each clock cycle, even if multiple BBs can be in flight simultaneously. Otherwise, the LSQ cannot determine which memory accesses among multiple simultaneously executing BBs should be allocated first. In order to support LSQs, Dynamatic conservatively keeps control flow sequential.

But sequential control flow can cause suboptimal performance. Figure 5.1 shows an example of a dynamically pipelined loop. The LSQ needs to allocate all the memory accesses triggered by the first input before starting to execute the second input. For this example, the allocation for the first input completes when the first input starts its last iteration. The second input can only enter the loop and start its first iteration after that time, even when there are no dependences between the inputs to the loop.

In the rest of Section 5.1, we show how to lift this restriction and enable C-slow pipelining in HLS for better throughput. C-Slow pipelining was first proposed by Leiserson *et al.* in 1983 [150]. It is a technique that enables multiple sets of data to be computed in the same

```
1 float a[N], b[N][M];
2 void triangleVecAccum() {
3   loop_0: for (int i = 0; i < N; i++) {
4     float s = a[f(i)];
5     loop_1: for (int j = 0; j < N-i; j++)
6       s = g(s, b[i][j]);
7     a[h(i)] = s;
8   }
9 }
```

Figure 5.2: Motivating example for dynamic C-slow pipelining.

hardware in the form of multiple threads. Compared to traditional pipelining, C-slow pipelined hardware executes in an out-of-order control flow.

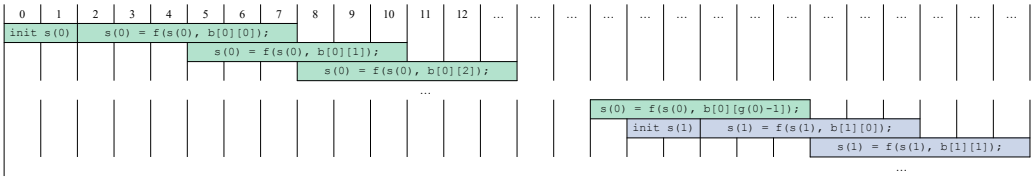
We tackle two problems in dynamically pipelining nested loops: 1) How can the restriction to sequential control flow be lifted to enable C-slow pipelining? 2) How can the memory correctness of C-slow pipelined hardware be preserved? Our main contributions are:

- a technique that statically determines a set of possible parallel memory-legal control flows for nested loops;
- a transformation pass that enables C-slow pipelining, which executes loop iterations early by inserting their schedules into empty pipeline slots of previous iterations; and
- analysis and results showing that over a set of benchmarks from Section 4.1.5, our approach achieves 1.75-5× speedup with only 1-20% area overhead.

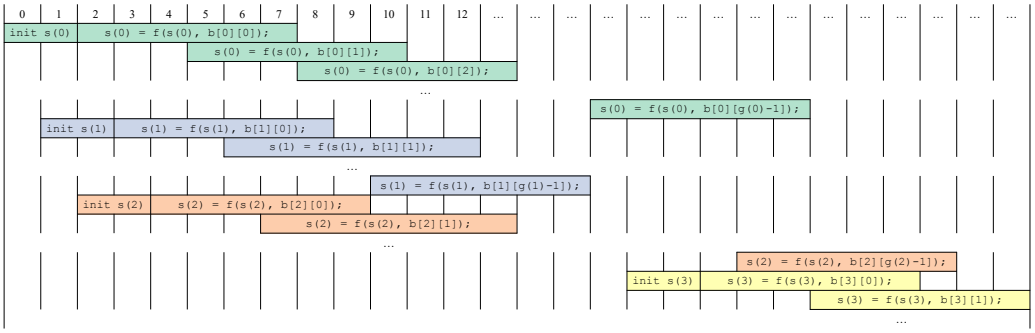
5.1.2 Motivating Example

In this section, we use a motivating example to demonstrate the problem of pipelining a nested loop. In Figure 5.2, a loop nest updates the elements in an array `a`. The outer loop `loop_0` loads an element at address `f(i)`. The inner loop `loop_1`, bounded by `N-i`, computes `s` with a row in an array `b`, shown as function `g`. The result is then stored back to array `a` at address `h(i)` at the end of the outer-loop iteration.

For simplicity, assume there is no inter-iteration dependence in the outer loop `loop_0`. Assume the latency of function `g` is 3 cycles. An inter-iteration dependence of the inner loop `loop_1` on



(a) Default pipeline schedule.



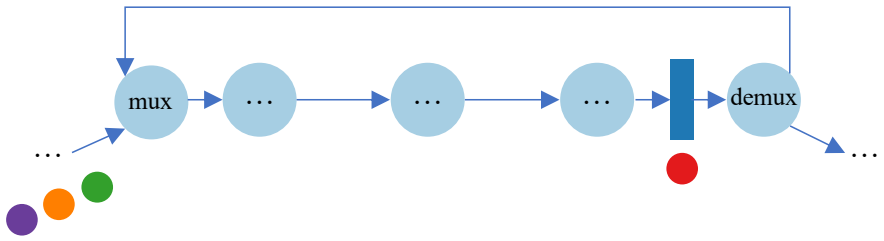
(b) Proposed pipeline schedule.

Figure 5.3: Schedules for the example in Figure 5.2. Assume there is no inter-iteration dependence in `loop_0`, and each instance of `loop_1` has a minimum II of 3. The default pipeline schedule only starts the second iteration of `loop_0` after the last iteration of `loop_1` in the first iteration of `loop_0` starts. Our approach inserts the following iterations of `loop_0` into the empty slots of its first iteration.

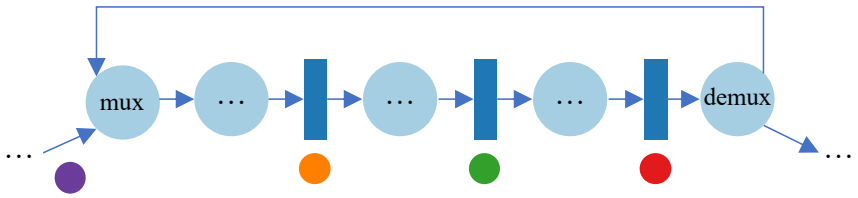
`s` causes a minimum II of 3. The pipeline schedule of the hardware from vanilla Dynamatic is shown in Figure 5.3a. The first iteration of `loop_0` is optimally pipelined with an II of 3 shown as the green bars. However, the second iteration, shown as blue bars, can only start after the last iteration of `loop_1` in the first iteration of `loop_0` starts.

The schedule shown in Figure 5.3b is also correct and achieves better performance. Since the loop II of `loop_0` is 3, the two empty slots between every two consecutive iterations allow the next two iterations of `loop_0` start earlier. The second iteration of `loop_0` now starts one cycle after the start of the first iteration of `loop_0`, followed by the third iteration shown as orange bars. After the last iteration of `loop_1` starts, the current inner-loop instance leaves new empty pipeline slots spare. This triggers the start of the fourth iteration, shown as yellow bars, filling into the new empty slot.

The reason that Dynamatic cannot achieve the schedule in Figure 5.3b is that the control flow in the latter schedule is out-of-order. The LSQ cannot retain the original program order of memory accesses and cannot verify the correctness of memory order from the out-of-order control flow, which may lead to wrong results. In this section, we use static analysis to prove



(a) Initial version of a loop.



(b) A 3-slowed loop.

Figure 5.4: An example of 3-slowing a loop. The 3-slowed loop has tripled latency, the same throughput and one-third critical path compared to the initial version.

such a control flow will still maintain a legal memory access order for a given program, so the LSQ still works correctly for this new program order.

This is an example where traditional techniques such as loop unroll and jam [151] may improve performance but also cause significant area overhead by duplicating the loop bodies. In this section, we propose a more area-efficient approach that achieves similar performance.

5.1.3 Background

C-slow pipelining is a technique that replaces each register in the circuit with C registers to construct C independent threads [152]. The circuit then operates as C-thread hardware while keeping one copy of resources. For instance, a stream of data enters a pipelined loop in Figure 5.4a. The loop computes with an II of 1, as illustrated by the presence of one register in the cycle. Assume the loop trip count is N, then the latency of the loop is approximately N cycles for a large N. The overall throughput of the hardware is 1/N and the critical path is the delay of the cycle. Assume that each set of data is independent from other sets. Figure 5.4b demonstrates a 3-slowed loop that is functionally equivalent to the one in Figure 5.4a. There are three registers in the cycle, evenly distributed in the path. This increases the latency of the hardware to 3N cycles. The loop can iterate with three sets of data in the cycle concurrently.

Then the overall throughput of the hardware is approximately 3/(3N) = 1/N, and the critical path is nearly 1/3 of the one in Figure 5.4a. A C-slowed loop can either have a better throughput or a better maximum clock frequency to achieve approximately C times speedup.

C-slow pipelining was first proposed by Leiserson *et al.* for optimising the critical path of synchronous circuits [150]. Markovskiy and Patel [152] propose a C-slow based-approach to improve the throughput of a microprocessor. Weaver *et al.* [153] propose an automated tool that applies C-slow retiming on a class of applications for certain FPGA families. Our work brings the idea of C-slow pipelining into the dynamic HLS world. We analyse nested loops at the source level to determine C for each loop by checking the dependence between inputs to the loop and then apply hardware transformations to achieve C-slow pipelining.

5.1.4 Methodology

In this section, we first formalise the restriction caused by LSQs in dynamic pipelining based on the formalisation given in Section 3.1.5. Second, we show how to determine a good C for C-slow pipelining using both dependence analysis and throughput analysis. Then we illustrate the design of our proposed loop scheduler. Finally, we demonstrate our tool flow integrated into the open-sourced HLS tool Dynamatic as a prototype.

Problem Formulation and Dependence Analysis

C-slow pipelining is amenable to improving the throughput when 1) a hardware design that has an II of greater than 1; and 2) it allows out-of-order execution of control flow. To simplify the problem, we restrict the scope of our work to nested loops. A loop in a CFG is defined as a set of consecutive BBs with back edges. Here we define the following terms for an inner loop in a loop nest:

- $B_L \subseteq B$ denotes the set of all the BBs in a loop,

- $E_L(j,k) \subseteq E_B$ denotes the BB executions in the kth iteration of the jth instance of the loop.

The II of each loop may vary between loop iterations as it is dynamically scheduled. For a loop, the maximum II of the loop at run time can be defined as:

$$\forall b,k.\, b \in B_L \wedge k > 1 \wedge (b,k) \in E_B \wedge (b,k-1) \in E_B \Rightarrow II \geq t(b,k) - t(b,k-1) \tag{5.4}$$

An II of greater than 1 means that there are empty pipeline slots in the schedule, which could be attributed to a lack of hardware resources or inter-iteration dependences. Here we assume infinite buffering and only analyse the case for the stall caused by inter-iteration dependences.

Constraint 5.2 forces each iteration of a loop to start execution sequentially, which can derive the following constraints for the loop:

$$\forall j,k,e,e'.\, e \in E_L(j,k-1) \wedge e' \in E_L(j,k) \Rightarrow t(e) < t(e') \tag{5.5}$$

$$\forall j,k,k',e,e'.\, e \in E_L(j-1,k) \wedge e' \in E_L(j,k') \Rightarrow t(e) < t(e') \tag{5.6}$$

Constraint 5.5 means that all the BB executions in an iteration of the loop cannot start unless all the BB executions in its last iteration have started. Constraint 5.6 means that all the BB executions in an instance of the loop cannot start unless all the BB executions in its last instance of the loop have started. The difference of the start times between two iterations when an II is greater than 1, also known as the empty pipeline slots, cannot be filled with the following iterations because of constraint 5.5 and constraint 5.6. If it is proven that the jth instance and the (j-1)th instance of the loop are independent, constraint 5.6 can be relaxed as the absence of dependence enables early execution of the jth instance. This allows the following iterations to start early, filling into the empty slots as illustrated in Figure 5.3b.

The parallelism among iterations depends on the number of consecutive independent instances, also known as the dependence distance of the outer loop. The minimum dependence distance

C of the outer loop of a loop l must satisfy the following:

$$d''(j, j') = (\exists k, k', e, e'.\, e \in E_L(j, k) \wedge e' \in E_L(j', k') \Rightarrow d(e, e')) \tag{5.7}$$

$$\forall j, c.\, j > C \wedge 1 \leq c \leq C \Rightarrow \neg d''(j - c, j) \tag{5.8}$$

The constraint for a C-slowed nested loop is that there are always *at most* C outer-loop iterations executing concurrently.

Although the outer-loop iterations are parallelised, the execution of inner loops still following constraint 5.5, where the inter-iteration dependences of the inner loops are respected. Constraint 5.6 transformed from constraint 5.2 is then relaxed to the following combined with constraint 5.5.

$$\forall j, k, e, e'.\, j > C \wedge e \in E_L(j - C, k) \wedge e' \in E_L(j, k) \Rightarrow t(e) < t(e') \tag{5.9}$$

Constraint 3.17 still holds as only the independent iterations execute earlier. The starting order of BB executions outside the C-slowed loop remains the same as vanilla Dynamatic. The starting order of BB executions inside the C-slowed loop is now in parallel and out-of-order.

The following sections explain how to solve two main problems: 1) How to efficiently determine C for a correct schedule with better performance? 2) How to transform the hardware to realise C-slow pipelining?

Exploration for C using Dependence Analysis

The dependence constraint above is equivalent to that the minimum dependence distance of the outer loop is less than C. A loop-carried data dependence always has a dependence distance of 1, therefore, we only need to analyse memory dependences. Our tool automatically generates a Boogie program to describe the memory behaviour of the nested loop and calls the Boogie verifier to prove the absence of memory dependence within a given distance. Microsoft Boogie [83] is an automated program verifier on top of satisfiability modulo theories (SMT) solvers. It has

its own intermediate verification language to describe the behaviour of a program to be verified, which can be automatically translated into SMT queries. An SMT solver then reasons about program behaviour including the values that its variables may take. Boogie is a verification language, which has its own constructs [83]. The ones used in this section are as follows:

1. `if (*) {A} else {B}` arbitrarily does `A` or `B`.
2. `havoc x` assigns an arbitrary value to a variable or an array `x`. It tells the verifier to prove the condition under all the possible values of `x`.
3. `assert c` proves the condition `c` for all the values that the variables in `c` may take.

For example, Figure 5.5 illustrates the Boogie program generated for the motivating example in Figure 5.3. It tries to prove the absence of memory dependence between any two outer-loop iterations with a distance of less than C, which mainly includes two parts. The Boogie procedure in Figure 5.5a arbitrarily picks a memory access from the nested loop during the whole execution and returns its parameters. The returned parameters include the label of the statement being executed, the array and address of the accessed memory, the iteration index of the outer loop and the type of the memory access. Detailed definitions of these parameters are listed in lines 6-11 in Figure 5.5b.

In Figure 5.5a, the `for` loop structures in Boogie are directly generated by the automated tool named EASY [88]. In the loop body, each memory access is replaced with an `if(*)` statement. The `if(*)` statement arbitrarily chooses to return the parameters of the current memory access or continue. The procedure is then able to capture *all* the memory access that may execute during the whole execution. If all these memory accesses are skipped, the procedure exits at line 22 with a false `valid` bit, indicating that the returned parameters are invalid.

Figure 5.5b describes the main Boogie procedure for dependence analysis. It takes a given C as an input. In line 4, it assumes that arrays `a` and `b` hold arbitrary values. This makes the verification results independent from the program inputs. Lines 12 and 13 arbitrarily pick two memory accesses from the nested loop using the procedure in Figure 5.5a.

```
procedure pickOneMemoryAccessFromLoop() returns (
valid: bool, stmt: int, addr: Index, array: Array,
iteration:Index, type: MemoryType) {
  loop_0: for (int i = 0; i < N; i++) {
    // s = a[f(i)];
    if (*) {
      valid := true; stmt := 0; addr := (f(i));
      array := a; iteration := (i); type := LOAD;
      return; }
    loop_1: for (int j = 0; j < g(i); j++)
      // s = f(s, b[i][j]);
      if (*) {
        valid := true; stmt := 1; addr := (i, j);
        array := b; iteration := (i); type := LOAD;
        return; }
    // a[h(i)] = s;
    if (*) {
      valid := true; stmt := 2; addr := (h(i));
      array := a; iteration := (i); type := STORE;
      return; }
  }
  valid := false;
  return;
}
```

(a) Procedure that arbitrarily picks a memory access.

```
// C : Given dependence distance
procedure main(C: int) {
  // assume that all the arrays have arbitrary values
  havoc a, b;

  // valid: whether the returned memory access is valid
  // stmt: which statement that executes the memory access
  // addr: which address the memory access touches
  // array: which array the memory access touches
  // iteration: the iteration index of the current outer-loops
  // type: the type of the memory access, either load or store
  call valid_0, stmt_0, addr_0, array_0, iteration_0, type_0 :=
    pickOneMemoryAccessFromLoop();
  call valid_1, stmt_1, addr_1, array_1, iteration_1, type_1 :=
    pickOneMemoryAccessFromLoop();

  assert !valid_0 || !valid_1 ||
         array_0 != array_1 ||
         stmt_0 == stmt_1 ||
         (type_0 == LOAD && type_1 == LOAD) ||
         iteration_0 >= iteration_1 ||
         getDistance(iteration_0, iteration_1) >= C ||
         addr_0 != addr_1;
}
```

(b) Main procedure that describes absent dependence for a given `C`.

Figure 5.5: Boogie program generated for the example in Figure 5.3. It tries to prove the absence of memory dependence between any two outer-loop iterations with a distance less than `C`.

The assertion at line 15 proves the dependence constraint as shown in constraint 5.8 for a given C. First, the two picked memory accesses from lines 12 and 13 must hold valid parameters (line 15). Second, two accesses touching different arrays cannot have a dependence (line 16). Two accesses executed by the same statement are safe (line 17), since the data flow to the same hardware operator must be in order as shown in constraint 5.2. Two loads cannot have dependence (line 18). Two returned memory accesses are arbitrary and have no difference. Here we assume the memory access with index 0 executes in an earlier outer-loop iteration than the one with index 1 (line 19). In the C-slow pipelining formulation, only the outer-loop iterations with an iteration distance less than C can execute concurrently (line 20). Any two memory accesses that exclude the cases in lines 15-20 cannot access the same address. The assertion must hold for *any* two memory accesses for any input values. The Boogie verifier automatically verifies whether the assertion always holds for a given C. If the assertion always holds, then it is safe to parallelize C iterations of the outer loop.

Besides the dependence constraints, a resource constraint of C-slow pipelining is that each path must be able to hold at least C sets of data. Dynamatic already inserts buffers into the hardware for high throughput. Our hardware transformation pass inserts an additional FIFO with a depth of C (named as C-slow buffers) in each control path cycle of the inner loop to adapt C-slow pipelining, which can hold at least C tokens.

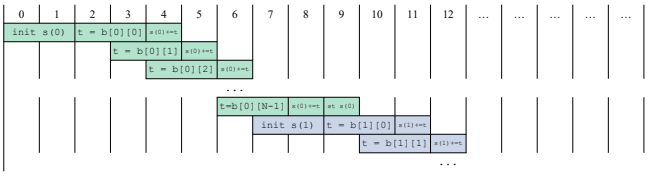
Exploration for C using Throughput Analysis

The dependence analysis above only defines a set of Cs that are suitable for C-slow pipelining. Now we show how to automatically determine an optimised C among these Cs using throughput analysis. The Boogie program determines an upper bound C that cannot break any dependence. However, a large C may not improve the overall throughput but only cause more area overhead.

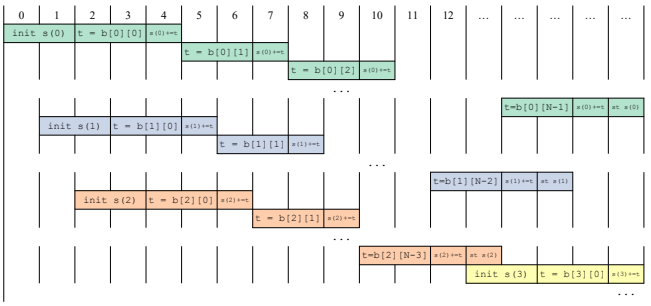
For example, Figure 5.6 illustrates an example where C-slow pipelining does not improve overall throughput. Figure 5.6a shows a function named `vecTrans` that transforms an array named `a`. The function contains a nested loop. In the outer loop, it loads the element in array `a` and accumulates the values in matrix `b` onto the element. In the inner loop, the elements in matrix

```
1  int a[N], b[N][M];
2
3  void vecTrans() {
4    loop_0:
5    for (int i = 0; i < N; i
       ++) {
6      int s = a[i];
7      loop_1:
8      for (int j = 0; j < N-i;
        j++) {
9        int t = b[i][j];
10       s += t;
11     }
12     a[i] = s;
13   }
14 }
```

(a) Source



(b) Default pipeline schedule.



(c) 3-slowed schedule.

Figure 5.6: An example where C-slow pipelining does not improve throughput. A C of 3 only leads to out-of-order execution and additional area.

`b` are accumulated in a triangle form.

The default pipeline schedule of function `vecTrans` is shown in Figure 5.6b. In the schedule, both the inner loop and the outer loop are fully pipelined. Although there is a carried dependence in the inner loop on the variable `s`, the integer adder has a latency of one clock cycle, leading to an II of 1.

This example is not amenable to C-slow pipelining, but this cannot be identified by Boogie. In the dependence distance analysis, Boogie found that C can be any positive integer as there is no inter-iteration dependence in the outer loop. For example, Figure 5.6c shows a 3-slowed schedule for function `vecTrans`. In the schedule, the start time of iterations in the first inner loop instance is delayed by three cycles, allowing two iterations in the second and third inner loop instances to start early. Such transformation preserves correctness but has no impact on the overall throughput. For this example, C-slow pipelining only causes more area overhead by adding the additional scheduler for out-of-order execution.

The condition where C-slow pipelining potentially improves the overall throughput is that the II of an inner loop is greater than 1. This leaves empty pipeline slots for early execution of the later inner loop instances. When the program is dynamically scheduled, the II of an inner loop may vary at run time. Here we use probabilistic analysis to statically infer an optimised

C from the average II.

Figure 5.7a shows a code example where the dependences in both the inner loop and the outer loop are dynamic. In the outer loop, the function loads and computes an element in array `a` at an index of `f(i)`. It then updates the element in the same array at an index of `h(i)`. In the inner loop, a variable `s` updates itself based on the elements in matrix `b`, which is used to update array `a` in the outer loop.

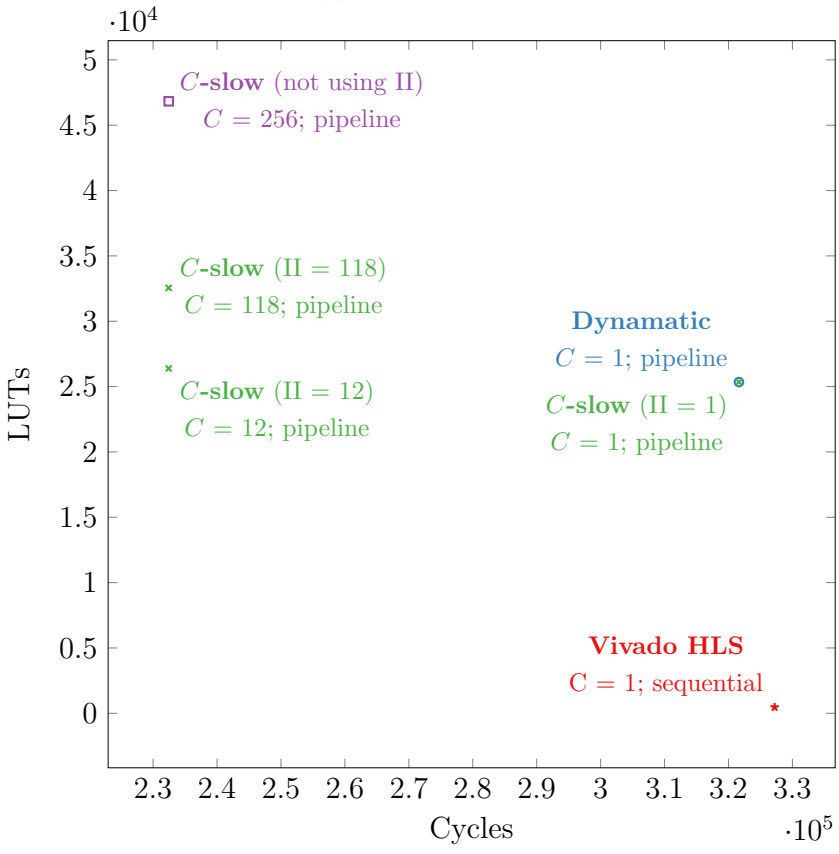
In the outer loop, the memory dependences between the loads and stores with array `a` are dynamic and depend on the iteration index of the out-loop `i`. For simplicity, assume that the minimum dependence distance in the outer loop for given `f(i)` and `g(i)` is no less than 256. That is, C-slow pipelining is valid for any C where $1 \leq C \leq 20$.

In the inner loop, the data-dependent condition causes two possible IIs. When the condition at line 11 is false, function `g` is skipped, leading to an II of 1. When the condition is true, function `g` is executed and the carried dependence on `s` causes an II of 118. The II of 118 is contributed by the latency between the input and the output of function `g`. The overall throughput then depends on the distribution of elements in matrix `b`, which affects the condition.

We then have 256 options to choose C no greater than 256, limited by the dependence in the outer loop. Here we discuss three approaches for choosing C based on the II of the loop. First, choosing a C based on an optimistic II reduces the area overhead of adding the scheduler for C-slow pipelining. However, a small C may limit the parallelism among inner loop instances when there are more empty slots for certain input data. For this example, an extreme case is where the minimum II is 1, which disables C-slow pipelining ($C = 1$ is equivalent to a single thread). Second, choosing a C based on a conservative II enables sufficient pipeline slots for early executing following inner loop instances. However, a large C may add unnecessary area overhead when there are only a small number or none of the empty pipeline slots, as illustrated in Fig, 5.6b. Finally, choosing a C based on an average II that may balance the area overhead and parallelism, achieving a more efficient hardware design. Our tool flow reuses the results of the throughput analysis during the Dynamatic synthesis flow. The buffering process in Dynamatic already uses static analysis to estimate the II of each control path.

```
1  float a[N], b[N][M];
2  void dynamicVecAccum() {
3    loop_0:
4    for (int i = 0; i < N; i++) {
5
6      int s = a[f(i)];
7
8      loop_1:
9      for (int j = 0; j < i; j++) {
10
11       // Dynamic carried dependence on s
12       // causes II = 1 or 118
13       if (b[i][j] != 0)
14         s = g(s, b[i][j]);
15
16     }
17
18     // Dynamic memory dependence distance
19     // causes dynamic II
20     // d >= 256
21     a[h(i)] = s;
22
23   }
24 }
```

(a) Source code.



(b) Area and performance using different approaches.

Figure 5.7: An example where both the inner loop and the outer loop have dynamic carried dependence, leading to dynamic IIs. Choosing C from average IIs achieves the best performance, and has the minimum area among dynamically scheduled hardware. The results are measured from uniformly distributed data.

Figure 5.7b shows the results of area and performance of the example in Figure 5.7a using different pipeline approaches. First, we evaluate three baselines: vanilla Dynamatic [13] for dynamic scheduling, Vivado HLS [14] for static scheduling, naive C-slow pipelining with only dependence constraints [154]. Vanilla Dynamatic allows pipelining loop with dynamic dependence, achieving better performance than static scheduling. However, the use of load-store queues that dynamically schedules memory operations causes significant area overhead. On the other hand, Vivado HLS that uses static scheduling cannot resolve data-dependent dependences at compile time, and keeps the loop sequential. The resultant hardware design sitting below vanilla Dynamatic has poor performance but high area efficiency because of resource sharing at compile time. Finally, the naive C-slow pipelining only analyses the dependence distance in the outer loop for choosing C. It generates hardware sitting on the top left that has better performance than both because it allows early execution of later inner loop instances. However, the value of C is over-approximated to a large value. This results in unnecessary area overhead since only a small portion of the inserted C-slow buffer in the control path is used.

We also evaluate the three approaches mentioned above for choosing an optimised C. First, the minimum II indicates that there is no empty slot in the inner loop schedule, preventing the transformation for C-slow pipelining. This leads to the same design as vanilla Dynamatic. The case where these two points overlap is only for this particular benchmark, where the minimum II of 1. Otherwise, the result with the minimum II greater than 1 may not overlap with the result by vanilla Dynamatic. Second, the maximum II indicates the best case that maximises parallelism for c-slow pipelining. However, this could still cause unnecessary area overhead when the iterations that have the maximum II are rare, such as in the case where only one iteration has the maximum II and the rest have the minimum II. Finally, the average II estimates the overall throughput of the inner loop. Such analysis reduces the C-slow buffer size while preserving sufficient slots for parallelism and maintaining the high performance of the naive C-slow approach. The constraint for an optimised C is then:

$$C \leq \min(C_D, II_{av}) \tag{5.10}$$

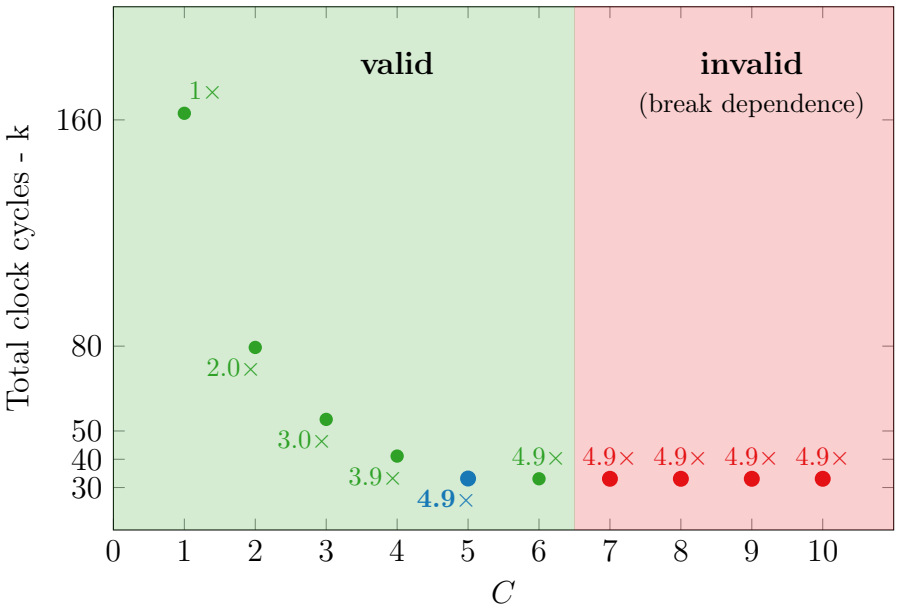


Figure 5.8: Speedup by varying C for the example in Figure 5.3. $C > 6$ breaks the memory dependence in the outer loop. Increasing C initial improves the throughput. However, once all the empty slots are filled, further increasing C has less effect on the throughput. Our tool automatically determines an optimal $C = 5$, shown in blue.

where C_D is the minimum C that passes the Boogie verification, and II_{av} is the average II of the inner loop.

Here we take the motivating example as a case study and then discuss the overall results for all the benchmarks. Figure 5.8 shows the total clock cycles of the hardware for the motivating example with different C. Only $C \leq 6$ for this example does not break memory dependence, where $C_D = 6$. When C increases initially, more outer-loop iterations are parallelized, significantly improving the throughput. The average II of the inner loop is 5. When C is greater than 5, the throughput remains the same since almost all the empty pipeline slots have been filled. The overhead caused by $C = 6$ only adds an additional depth to the FIFOs.

Hardware Transformation

Once the value of C for a loop is determined, our tool flow inserts a component named *loop scheduler* between the entry and exit of each loop. Each C is used as a parameter of the corresponding loop scheduler. The loop scheduler dynamically schedules the control flow and ensures that at most C iterations can execute concurrently. Any outermost loop or unverified

loop has its loop scheduler holding $C = 1$ and executes control flow sequentially.

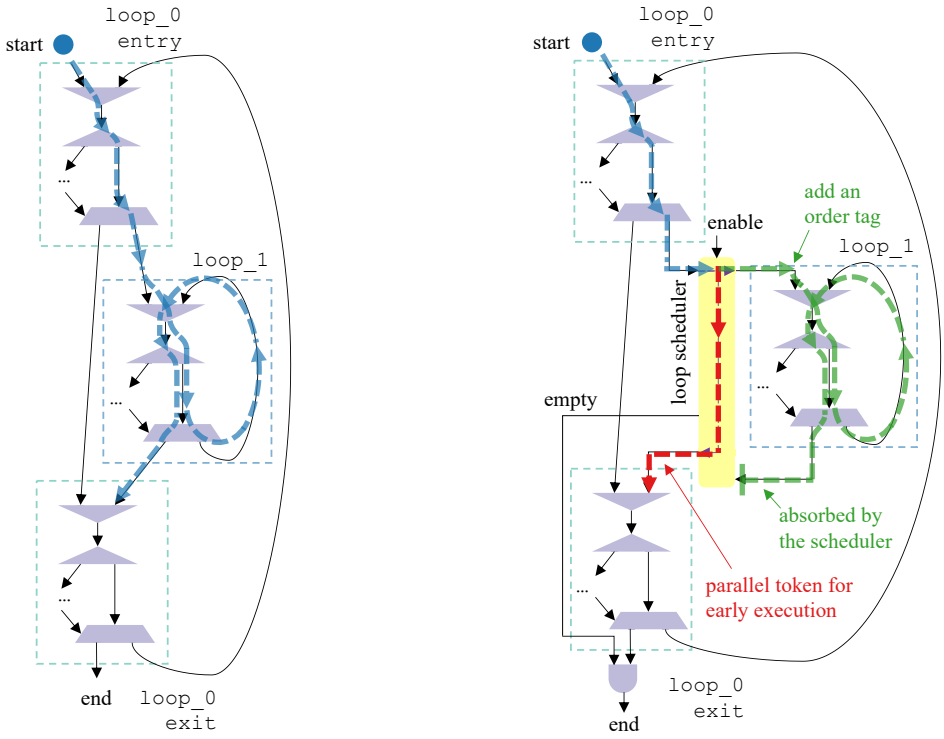
Figure 5.9 shows the proposed loop scheduler integrated into a dynamically scheduled control flow graph. For example, the control flow graph of the code in Figure 5.3 from vanilla Dynamatic is shown in Figure 5.9a. Each dotted block represents a BB. The top BB represents the entry control of the outer loop, which starts the outer iteration and decides whether to execute the inner loop. The middle BB represents the control of the inner loop. The bottom BB represents the exit control of the outer loop, which decides whether to exit the outer loop.

In each BB, a merge is used to accept a control token that triggers the start of the current BB execution. Then a fork is used to produce other tokens to trigger all the data operations inside this BB, hidden in the ellipsis. The control token flows through the fork to a branch. The branch decides the next BB to trigger based on the BB condition. The control flow in Figure 5.9a follows the following steps:

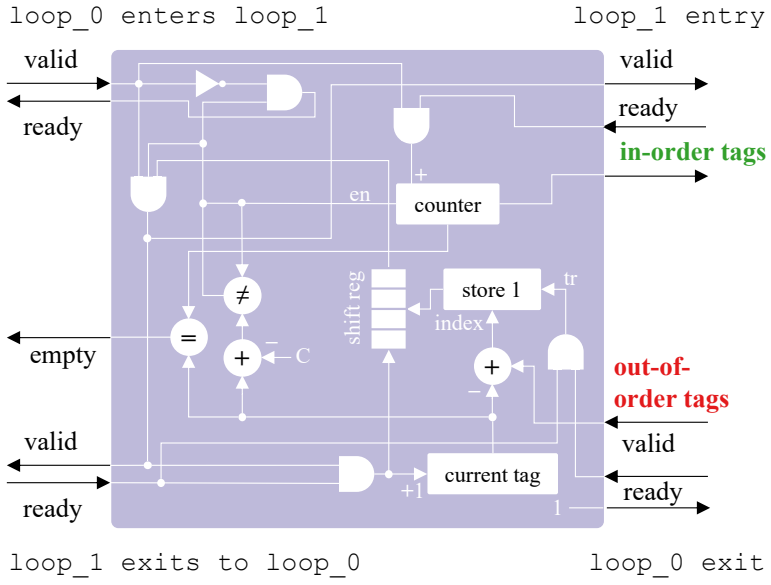
1. A control token enters the top BB to start;
2. The token goes through the top BB and enters the middle BB to start the inner loop.
3. The token circulates in the middle BB through the back edge until the exit condition is met.
4. The token exits the middle BB and enters the bottom BB. It either goes back to the top BB to repeat 2) or exits, depending on the exit condition.

The control flow is sequential as there is always at most one control token in the control path. Figure 5.9b shows the control flow graph with the proposed loop scheduler integrated into the inner loop. The loop scheduler for the outer loop has C of 1 and is neglected for simplicity. The control flow is then:

1. A control token t_1 enters the top BB to start.
2. t_1 goes through the top BB and enters the middle BB with a tag added by the loop scheduler. The loop scheduler checks if there are fewer than C(=3 in this example)



(a) Default CFG. (b) Transformed CFG.



(c) Loop scheduler

Figure 5.9: Our tool flow considers each instance of the innermost loop as a thread and achieves the schedule in Figure 5.3b. The dashed arrows represent the token transition in control flow. The scheduler tags the control tokens in the innermost loop to reorder them at the output after out-of-order execution.

tokens in the inner loop. If yes, it immediately produces another token t_2 and sends it to the bottom BB to execute the control flow early (indicated as the red dashed arrow).

3. t_1 circulates in the middle BB. t_2 goes to the top BB and enters the middle BB with another tag. The loop scheduler produces another token t_3 and sends it to the bottom BB.

4. The above repeats and t_4 is produced. t_1, t_2 and t_3 are all circulating in the middle BB.

5. t_4 reaches the branch in the top BB but is *blocked* by the loop scheduler until one token exits the middle BB and is consumed by the loop scheduler.

6. The above repeats until the last token exits the bottom BB. An AND gate is inserted at the exit of the bottom BB to synchronise the control flow. It requires a token from the exit of the nested loop and there is no token remaining in the inner loop.

Since the execution order of loop iterations has changed, all the data flows must be strictly scheduled by the control flow. Dynamatic uses merges to accept input data at the input of a BB when there is always at most one valid input. There may be multiple valid inputs at the start of BB after the transformation. In order to preserve correctness, we replace all the merges in the inner loop with muxes, such that the data is always synchronised with the control token and can recover in-order using the tag in the control token. An advantage of this approach is that only the control tokens need to be tagged to preserve the original order, where the data flow is always synchronised by the control flow.

The design of the loop scheduler is shown in Figure 5.9c. It guards the entry and exit of the inner loop. The ready signal at the bottom right is forced to 1 as the scheduler already controls the input throughput using C and there cannot be back pressure at the output. In the scheduler, a counter is used to count the number of executing control tokens in the inner loop. Based on the value of the counter and the specified C, it decides whether to accept the token from the outer loop and replicates a token to the output to the outer loop for early execution of the next outer-loop iteration. The input token from the exit of the inner loop decrements

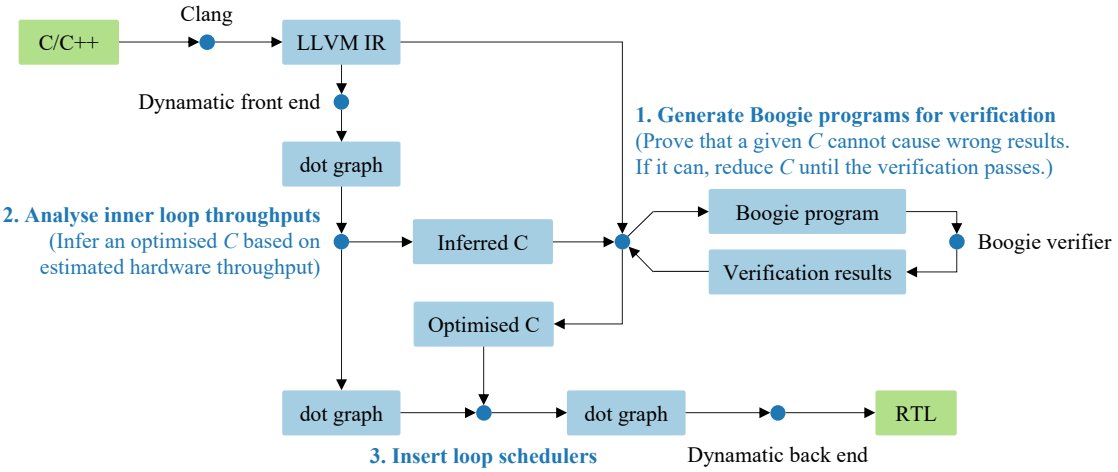


Figure 5.10: Our work integrated into Dynamatic. Our contributions are highlighted in bold blue text.

the counter value by one, allowing the next token from the outer loop to enter the inner loop. An empty bit is set to indicate that there is no control token in the inner loop.

Tool Flow

We integrate our work into Dynamatic for prototyping, as shown in Figure 5.10. First, the input C/C++ program is lowered into LLVM IR. A control data flow graph is then generated in the form of a dot graph by the front end of Dynamatic. Based on the existing throughput analysis report in Dynamatic, our analyser determines a starting C for each innermost loop in the input program. Then a Boogie program is generated for dependence analysis and checked by the Boogie verifier. Our analysis in LLVM passes searches for a maximum valid C no greater than the starting value. Once C is determined, our transformation pass inserts loop schedulers and FIFOs into the dot graph. Finally, the transformed graph is translated to RTL code by the back-end of Dynamatic as the final design.

5.1.5 Experiments

We evaluate our work on a set of benchmarks, comparing with the designs using Xilinx Vivado HLS and Dynamatic in total circuit area and wall clock time. The total clock cycles were

obtained using Vivado XSIM simulator, and the area results were obtained from the post P&R report in Vivado. The FPGA device we used for result measurements is xc7z020clg484. The version of Xilinx software is 2019.2.

Results

We first take the motivating example as a case study and then discuss the overall results for all the benchmarks. Figure 5.8 shows the total clock cycles of the hardware for the motivating example with different C. Only $C < 6$ for this example does not break memory dependence. When C increases initially, more outer-loop iterations are parallelized, significantly improving the throughput. However, when C reaches 5, further increasing C does not affect the throughput. That is because almost all the empty pipeline slots have been filled. The overhead caused by $C = 6$ only adds an additional depth to the FIFOs. With a small C, the area overhead caused by the FIFO depths is neglectable for most applications.

The speedups of our designs compared to the baselines over all the benchmarks are shown in Figure 5.1, and the detailed results are shown in Table 5.2. Overall, the circuits from vanilla Dynamatic are slightly slower than Vivado HLS even though they have better throughput. It is because the academic buffering tool in Dynamatic cannot perform retiming as well as the commercialised Vivado HLS. This leads to a lower maximum clock frequency, especially for `covariance` and `gramSchmidt`. C-slow pipelining significantly improves the throughput of dynamically scheduled circuits, leading to the best performance in the figure for most benchmarks.

The actual speedup is less than the expected speedup based on the value C due to resource constraints. For instance, there are multiple memory accesses to the same array in one iteration, and the memory blocks have limited bandwidth. These memory accesses are then stalled and serialised by the internal memory arbiter, resulting in additional pipeline stalls. Such performance overhead could be reduced when array partitioning is applied. Even though, our tool achieves at least 1.75× speedup from vanilla Dynamatic. Additionally, benchmarks such as `correlation` and `gramSchmidt` contain a only small region of code that can be C-slow pipelined, resulting in less performance improvement. The maximum clock frequency decreases

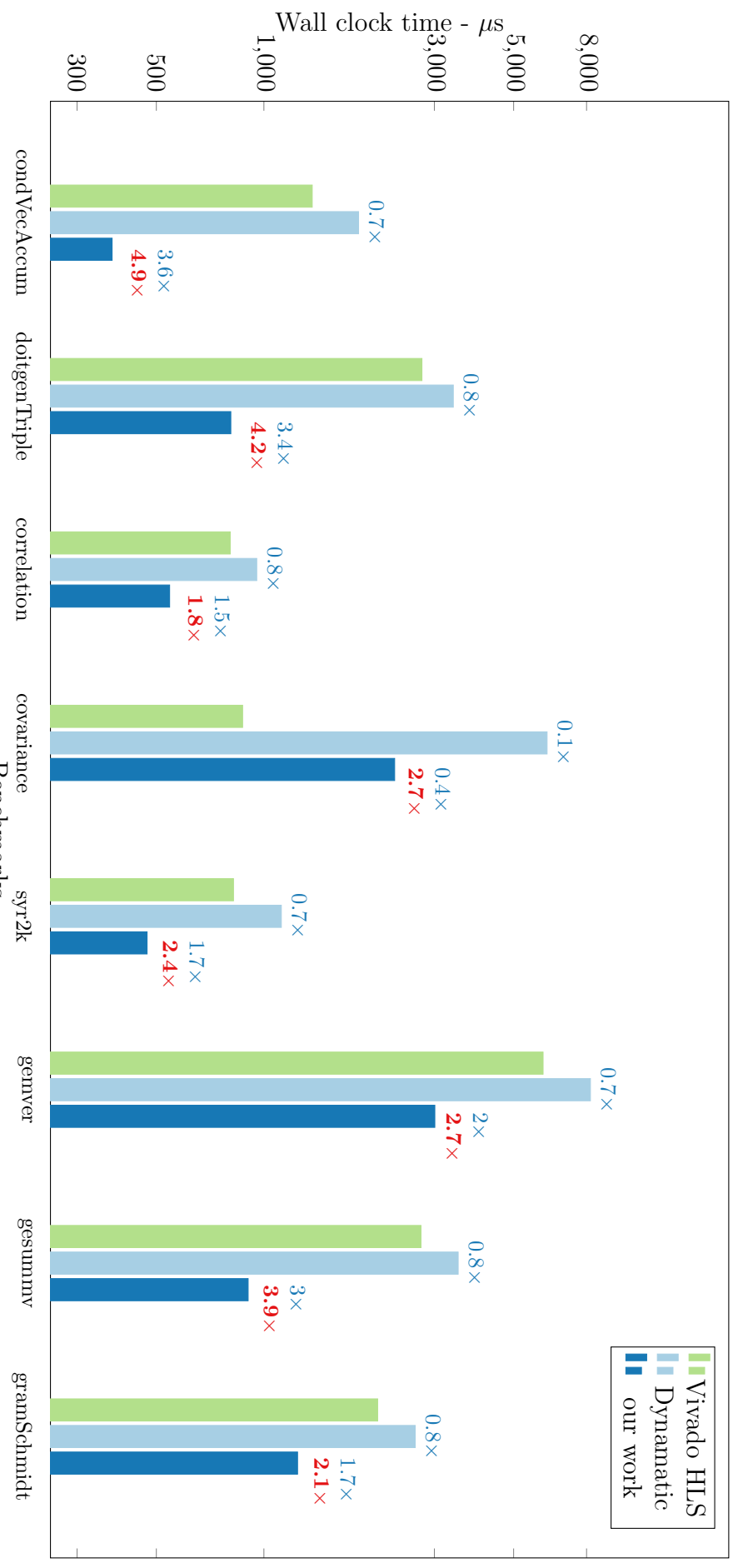


Table 5.1: Performance comparison of the designs from our tool with the designs from Vivado HLS and Dynamatic. The speedups shown in blue and red take the design from Vivado HLS and Dynamatic as the baselines respectively.

Table 5.2: Evaluation of our approach on a set of benchmarks. vhls = Vivado HLS. base = vanilla Dynamatic. ours = our work.

Benchmarks	LUTs - k				DSPs				Registers - k				Cycles - k				Fmax - MHz				Wall clock time - ms			
	vhls	base	ours	×	vhls	base	ours	×	vhls	base	ours	×	vhls	base	ours	×	vhls	base	ours	×	vhls	base	ours	×
triangleVecAccum	0.311	18.8	19.4	1.03	5	5	5	1	0.295	4.92	5.57	1.13	165	162	33.2	0.20	121	88.2	88.3	1	1.37	1.84	0.376	0.20
doitgenTriple	0.728	21.8	22	1.01	5	20	20	1	0.715	8.18	8.47	1.04	335	297	67.3	0.23	121	87.6	83.3	0.95	2.77	3.39	0.808	0.24
correlation	1.94	14.6	14.8	1.01	5	36	36	1	1.78	11.8	11.8	1	97.2	90.1	41.7	0.46	121	94.4	76.5	0.81	0.805	0.955	0.545	0.57
covariance	1.55	7.44	8.96	1.20	5	9	9	1	1.25	6.83	9.4	1.38	105	89.8	33.6	0.37	121	14.5	14.5	1	0.872	6.2	2.32	0.37
syr2k	0.603	4.04	4.57	1.13	5	19	19	1	0.693	3.97	4.44	1.12	99.3	82.4	34.3	0.42	121	73.6	72.9	0.99	0.822	1.12	0.471	0.42
gemver	1.44	8.33	8.72	1.05	5	28	28	1	1.24	8.4	8.29	0.99	730	719	227	0.32	121	87.7	75.3	0.86	6.04	8.2	3.01	0.37
gesummv	0.612	3.48	3.76	1.08	5	18	18	1	0.701	3.66	4.18	1.14	333	327	81.9	0.25	121	93.5	90.6	0.97	2.75	3.5	0.903	0.26
gramSchmidt	1.66	47.4	47.9	1.01	5	30	30	1	1.58	16.3	16.9	1.04	238	142	57.9	0.41	114	53.6	46.6	0.87	2.08	2.65	1.24	0.47
Geom. mean			**1.07**				1				**1.1**				**0.32**				**0.94**					**0.35**

slightly after C-slow pipelining because of the insertion of non-transparent FIFOs (latency = 0 cycle) increases the critical path of the circuit.

The area overhead in LUTs and registers is caused by the additional loop schedulers and FIFOs. DSPs are not changed as they are used for floating-point operators. The area overhead caused by our approach is relatively small compared to the performance gain.

5.1.6 Summary

Existing dynamically scheduled HLS tools require all BBs to start in strict program order, in order to respect any inter-BB dependences, regardless of whether dependences are actually present. This leads to missed opportunities for performance improvements by having BBs start simultaneously.

We propose an automated approach to lifting this restriction. We show how to statically identify sequences of consecutive subgraphs in the CFG of a program and reschedule them (with the help of the Microsoft Boogie verifier) to start simultaneously. We then show how to map the optimised schedule into efficient hardware designs. The performance gain is significant, while the area overhead is negligible.

5.2 Dynamic Inter-Block Scheduling

This section demonstrates another application of the proposed dependence model for achieving simultaneous execution of multiple independent BBs by parallelising independent sequential loops.

In Section 5.2.1, we explain the motivation and challenges of enabling inter-BB parallelism. In Section 5.2.2, we demonstrate a motivating example of parallelising independent sequential loops. In Section 5.2.3, In Section 5.2.4, we show how to use Microsoft Boogie to automatically determine the absence of dependence between BBs and how to use the analysis for parallelising

```
1  // f(i) = 0;
2  // g(j) = j*j+1;
3  // h(j) = j;
4
5  int a[N], b[M];
6  void transformVector() {
7    loop_0: for (int i = 0; i < X; i++)
8      b[i] = op0(a[f(i)];
9    loop_1: for (int j = 0; j < Y; j++)
10     a[g(j)] = op1(a[h(j)]);
11 }
```

Figure 5.11: Motivating example for inter-block scheduling.

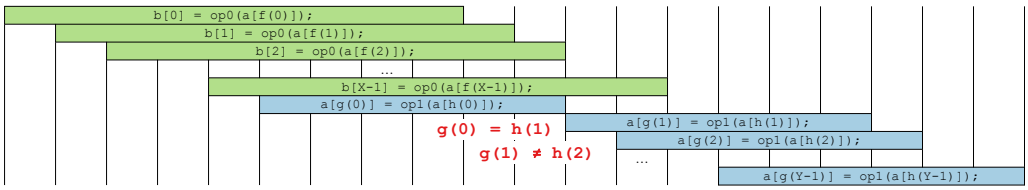
sequential loops. In Section 5.2.5, we evaluate the effectiveness of our approach on a set of benchmarks.

5.2.1 Introduction

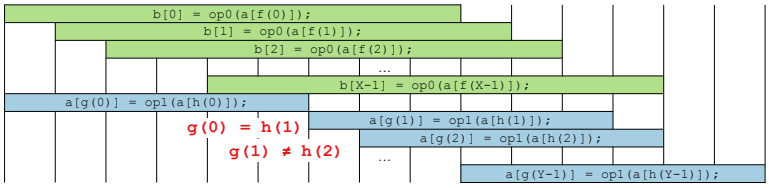
In this section, we focus on the inter-BB dependences as explained in Section 3.1.4. We find BBs that can be started out-of-order (or even simultaneously), and use static analysis (powered by the Microsoft Boogie verification engine [83]) to ensure that inter-BB dependences are still respected. We tackle two problems: 1) How to automatically identify BBs that can safely start in parallel? 2) How to synthesise efficient hardware that can start BBs in parallel? Our main contributions include:

- a technique that automatically identifies sequences of consecutive subgraphs from the control-flow graph (CFG) of a sequential program and reschedules these subgraphs for parallelism using the Microsoft Boogie verifier;
- a transformation pass that efficiently parallelises these subgraphs in hardware; and
- results and analysis showing that our approach, compared to the original Dynamatic, achieves 1.05-9.05× speedup (and 1.05-20.6× speedup when combined with our work on C-slow pipelining in Section 5.1), and almost the same circuit area.

5.2.2 Motivating Example



(a) Default pipeline schedule.



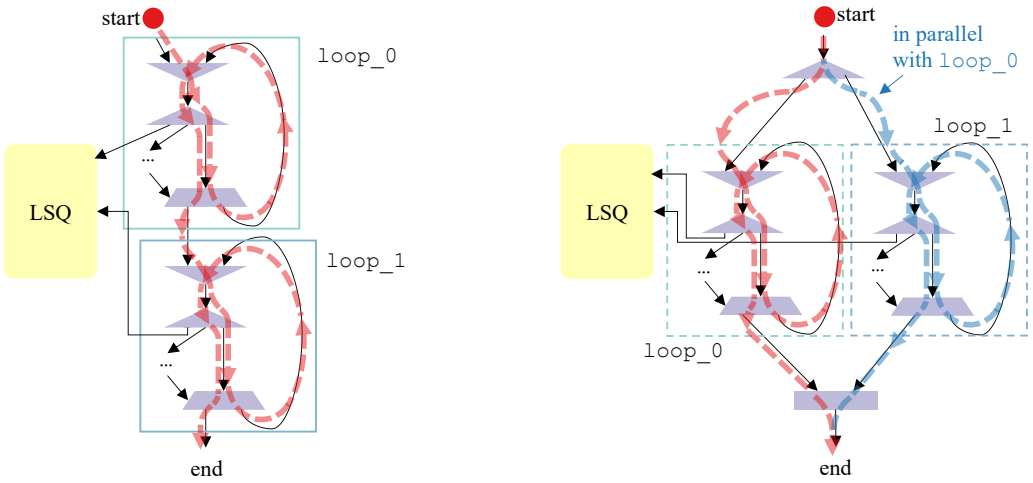
(b) Parallelised pipeline schedule.

Figure 5.12: Schedules generated by the example in Figure 5.11. Assume no dependence between two loops. The dynamically scheduled hardware from the original Dynamatic [13] a schedule in (b). Our work achieves an optimised schedule in (c).

Here we illustrate a motivating example of parallelising two sequential loops in dynamically scheduled hardware. Figure 5.11 shows an example of two sequential loops, `loop_0` and `loop_1`. In each iteration of `loop_0`, an element at index `f(i)` of array `a` is loaded and processed by a function `op0`. The result is stored to an element at index `i` of array `b`. In each iteration of `loop_1`, an element at index `h(j)` of array `a` is loaded and processed by a function `op1`. The result is stored back to array `a` at index `g(j)`. For simplicity, let `f(i) = 0`, `g(j) = j*j+1` and `h(j) = j`. Hence, there is no memory dependence between two loops, that is, $\forall 0 \leq i < \mathtt{X}. \forall 0 \leq j < \mathtt{Y}. \mathtt{f}(i) \neq \mathtt{g}(j)$.

Dynamatic [13] synthesises hardware that computes in a schedule shown in Figure 5.12a. The green bars represent the pipeline schedule of `loop_0`, and the blue bars represent the pipeline schedule of `loop_1`. In `loop_1`, the interval between the starts of consecutive iterations, known as the *initiation interval* (II), is variable because of the dynamic inter-iteration dependence between loading from `a[h(j)]` and storing to `a[g(j)]`. For instance, if we suppose that `g` and `h` are defined such that $\mathtt{g}(0) = \mathtt{h}(1)$, then the first two iterations must be executed sequentially, and if we further suppose that $\mathtt{g}(1) \neq \mathtt{h}(2)$, then the second and third iterations are pipelined with an II of 1.

However, `loop_1` is stalled until all the iterations in `loop_0` have started even though it has no dependence on `loop_0`. The reason is that Dynamatic forces all the BBs to start sequentially



(a) Default CFG. (b) Parallelised CFG.

Figure 5.13: Hardware transformation of the motivating example in Fig. 5.12.

to preserve any potential inter-BB dependence, such as the inter-iteration memory dependence in `loop_1`. For this example, each loop iteration is a single BB, and at most one loop iteration starts in each clock cycle.

An optimised schedule is shown in Figure 5.12b. In the figure, both loops start from the first cycle and iterate in parallel, resulting in better performance. Existing approaches cannot achieve the optimised schedule: static scheduling can start `loop_0` and `loop_1` simultaneously such as using multi-threading in LegUp HLS [31], but `loop_1` is sequential as the static scheduler assumes the worst case of dependence and timing; dynamic scheduling has a better throughput of `loop_1`, but cannot start it simultaneously with `loop_0`.

Besides, determining the absence of dependence between these two loops for complex `f(i)`, `g(j)` and `h(j)` is challenging. In this section, our tool flow 1) generates a Boogie program to formally prove that starting `loop_0` and `loop_1` simultaneously cannot break memory dependence and 2) parallelises these loops in dynamically scheduled hardware if they are proven independent. The Boogie program generated for this example is explained later (in Figure 5.14).

The transformation for the example in Figure 5.11 is demonstrated in Figure 5.13a and Figure 5.13b. Figure 5.13a shows the CFG generated by the original Dynamatic. The CFG consists of a set of pre-defined components, as listed in Table 3.1. As indicated by the red arrows, a control token enters the upper block and triggers all the operations in the first iteration of

`loop_0`. It circulates within the upper block for `X` cycles and then enters the lower block to start `loop_1`. Figure 5.13b shows a parallelised CFG by our tool flow. Initially, a control token is forked into two tokens. These two tokens simultaneously trigger `loop_0` and `loop_1`. A join is used to synchronise the two tokens when they exit these loops. Both designs use the same hardware. Figure 5.13b uses these resources in a more efficient way by allowing the two loops to be used in parallel, reducing the overall execution time. The rest of Section 5.2 explains the details of our approach.

5.2.3 Background

Dependence analysis for parallelising a CFG of a sequential program has been well-studied in the software compiler world [34]. Traditional approaches exploit BB parallelism using polyhedral analysers such as Pluto [155] and Polly [156]. These tools automatically parallelise code that contains affine memory accesses [157, 158] and have been widely used in HLS to parallelise hardware kernels [71, 74, 75, 159]. However, polyhedral analysis is not applicable when analysing irregular memory patterns such as non-affine memory accesses, which are commonly seen in applications amenable for dynamic scheduling, such as tumour detection [98] and video rendering [113].

Recently, there are works that use formal verification to prove the absence of dependence to exploit hardware parallelism. Zhou *et al.* [87] propose a satisfiability-modulo theory (SMT)-based [80] approach to verify the absence of memory contention in banked memory among parallel kernels. Cheng *et al.* propose a Boogie-based approach for simplifying memory arbitration for multi-threaded hardware [88].

Mapping a parallel BB schedule into hardware has also been widely studied. Initial work by Cabrera *et al.* [160] proposes an OpenMP extension to off-load computation to an FPGA. Leow *et al.* [161] propose a framework that maps OpenMP code in Handel-C [58] to VHDL programs. Choi *et al.* [162] propose a plugin that synthesises both OpenMP and Pthreads C programs into multi-threaded hardware, used in an open-sourced HLS tool named LegUp [31].

Gupta *et al.* propose an HLS tool named SPARK that parallelises control flow with speculation [163]. Existing commercialised HLS tools [14, 16, 17, 164] support multi-threaded hardware synthesis using manually annotated directives by users. These works either require user annotation or only use static scheduling, while our approach only uses automated dynamic scheduling.

Finally, there are works on simultaneously starting BB in dynamic-scheduling HLS. Section 3.2 proposes an HLS tool named DASS that allows each statically scheduled component to act as a static island in a dynamically scheduled circuit. Each island is still statically scheduled, while our tool flow only uses dynamic scheduling.

5.2.4 Methodology

Here we first show how to formalise the problem based on the model in Section 3.1.4. We then show how to extract sets of subgraphs from a sequential program, where subgraphs in the same set may start in parallel. The absence of dependence between these parallelised subgraphs is formally verified using the generated Boogie program by our tool.

Problem Formulation and Dependence Analysis

The search space for BBs that can start in parallel could be huge, and it scales exponentially with the code size. In order to increase scalability, we limit our scope to loops. Each loop forms a subgraph in the CFG for analysis. Parallelising BBs outside any loop adds significant search time but has negligible improvement in latency. We define the following terms:

- $G = \{g_1, g_2, ...\}$ denotes a set of consecutive subgraphs in the CFG of the program,
- $E_G \subseteq G \times \mathbb{N}$ denotes the executions of subgraphs,
- $\prec_G \subseteq E_G \times E_G$ denotes the original program order of subgraph execution, and
- $B_g \subseteq B$ denotes the set of all the BBs in subgraph g.

The subgraph set G must satisfy the following constraints based on the original sequential program order. First, the BB sets of all the subgraphs in G must be disjoint.

$$\forall g, g'.\, g \in G \wedge g' \in G \wedge g \neq g' \Rightarrow B_g \cap B_{g'} = \emptyset \tag{5.11}$$

Second, no BB outside a subgraph executes during the execution of the subgraph in sequential program order. Let $b_0(e)$ and $b_{\mathrm{n}}(e)$ denote the first BB execution and the last BB execution in a subgraph execution, where $e \in E_G$.

$$\nexists b, k, g, m.\, (g, m) \in E_G \wedge b \notin B_g \wedge (b, k) \in E_B \Rightarrow b_0(g, m) \prec (b, k) \prec b_{\mathrm{n}}(g, m) \tag{5.12}$$

Finally, all the subgraphs in G must consecutively execute, *i.e.* directly connected to at least one subgraph in CFG. That means that they are sequentially executed in each iteration. Let $c(g, g')$ denote whether the execution of subgraphs g' is consecutively after the execution of subgraph g.

$$c(g, g') = (\nexists m, g''.\, (g, m) \in E_G \wedge (g', m) \in E_G \wedge (g'', m) \in E_G \Rightarrow (g, m) \prec (g'', m) \prec (g', m)) \tag{5.13}$$

$$\nexists g, g', m, b, k.\, (g, m) \in E_G \wedge (g', m) \in E_G \wedge c(g, g') \wedge (b, k) \in E_B \Rightarrow b_{\mathrm{n}}(g, m) \prec (b, k) \prec b_0(g', m) \tag{5.14}$$

Now we start to map the execution of subgraphs into a hardware schedule. As explained in constraint 5.2 in Section 3.1.4, Dynamatic forces BB to start sequentially. This leads to the following constraint regardless any dependence.

$$\forall g, g', m.\, c(g, g') \wedge (g, m) \prec (g', m) \Rightarrow t(b_{\mathrm{n}}(g, m)) < t(b_0(g', m)) \tag{5.15}$$

If it is proven that the execution (g, m) cannot have dependence with the execution (g', m) of its consecutively following subgraph, there is no need to use muxes and LSQs to resolve the dependence between these two subgraphs. Then (g', m) can start execution early, such

as $t(b_0(g,m)) = t(b_0(g',m))$, which leads to a correct schedule with better performance. Let $d'(g,g')$ denote that two subgraphs g and g' may have dependence, and let $d'_c(g,g')$ denote that there exists a subgraph between the execution of g and g' in the same iteration including g that may have dependence with subgraph g'.

$$d'(g,g') = (\exists b,b'.\, b \in B_g \wedge b' \in B_{g'} \Rightarrow d'(b,b')) \tag{5.16}$$

$$d'_c(g,g') = d'(g,g') \vee (\exists g'',m.\, (g,m) \prec (g'',m) \prec (g',m) \Rightarrow d'(g'',g')) \tag{5.17}$$

Constraint 5.2 is now relaxed to:

$$\forall g,g',m.\, (g,m) \prec (g',m) \wedge d'_c(g,g') \Rightarrow t(b_{\mathrm{n}}(g,m)) < t(b_0(g',m)) \tag{5.18}$$

$$\forall b,b',k,k',g.\, b \in B_g \wedge b' \in B_g \wedge g \in G \wedge (b,k) \prec (b',k') \Rightarrow t(b,k) < t(b',k') \tag{5.19}$$

Constraint 5.18 restricts the starting time of a subgraph execution by its most recently executed subgraph that has dependences. Inside each subgraph, BB execution remains to start sequentially, as shown in constraint 5.19.

The optimised schedule still respects all the inter-BB dependences since it only modifies the start times of independent subgraph execution. The intra-BB dependences remain unchanged since the transformation is applied at only the subgraph level. Therefore constraint 3.17 still holds for the optimised schedule.

The following sections explain how to solve two main problems: 1) How to efficiently determine a large set of G and a highly parallelised schedule for G? 2) How to map a parallelised schedule into efficient hardware?

Subgraph Extraction

Given an input program, our tool flow analyses sequential loops in each depth and constructs a number of sets of subgraphs. Each set contains several consecutive sequential loops at the same depth, where each loop forms a subgraph. For instance, the example in Figure 5.12 has a set of

```
1  procedure pickOneMemoryAccess() returns (valid: bool,
2  addr: Index, array: Array, subgraphID: Index,
3  type: MemoryType) {
4    loop_0: for (i = 0; i < X; i++) {
5      // b[i] = op0(a[f(i)]);
6      if (*) { valid := true; addr := f(i); array := a;
7               subgraphID := 0; type := LOAD; return; }
8      if (*) { valid := true; addr := i; array := b;
9               subgraphID := 0; type := STORE; return; } }
10   loop_1: for (j = 0; j < Y; j++) {
11     // a[g(i)] = op1(a[h(j)]);
12     if (*) { valid := true; addr := h(j); array := a;
13              subgraphID := 1; type := LOAD; return; }
14     if (*) { valid := true; addr := g(j); array := a;
15              subgraphID := 1; type := STORE; return; } }
16   valid := false; return; }
```

(a) Procedure that arbitrarily picks a memory access.

```
1  procedure main() {
2    // assume that all the arrays have arbitrary values
3    havoc a, b;
4    // valid: whether the returned memory access is valid
5    // addr: which address the memory access touches
6    // array: which array the memory access touches
7    // subgraphID: which subgraph the memory access is in
8    // type: the type of memory access, either load/store
9    call valid_0, addr_0, array_0, subgraphID_0, type_0 :=
       pickOneMemoryAccess();
10   call valid_1, addr_1, array_1, subgraphID_1, type_1 :=
       pickOneMemoryAccess();
11   assert !valid_0 || !valid_1 ||
12          subgraphID_0 == subgraphID_1 ||
13          array_0 != array_1 ||
14          (type_0 == LOAD && type_1 == LOAD) ||
15          addr_0 != addr_1;
16 }
```

(b) Main procedure that proves the absence of dependence.

Figure 5.14: A Boogie program generated for the example in Figure 5.12. It tries to prove the absence of memory dependence between two sequential loops `loop_0` and `loop_1`.

two subgraphs, corresponding to `loop_0` and `loop_1`. Our tool flow then checks the dependence among the subgraphs for each set. Dynamatic translates data dependence into handshake connections in hardware for correctness. Our tool flow does not change these connections so the data dependence is still preserved. For memory dependences, our tool flow generates a Boogie program to prove the absence of dependence among subgraphs. For this example, Boogie proves that the two loops do not conflict on any memory locations and therefore can be safely reordered.

For example, Figure 5.14 shows the Boogie program that proves the absence of a dependence between `loop_0` and `loop_1` in Figure 5.12. The Boogie program consists of two procedures. First, the procedure in Figure 5.14a describes the behaviour of function `transformVector`

and arbitrarily picks a memory access during the whole execution. The procedure returns a few parameters for analysis, as listed in lines 1-3 in Figure 5.14b. The `for` loop structures are automatically translated using an open-sourced tool named EASY [88]. In the rest of the function, each memory operation is translated to a non-deterministic choice `if(*)`. It arbitrarily returns the parameters of a memory operation or continues the program. If all the memory operations are skipped, the procedure returns an invalid state in line 16. The non-deterministic choices over-approximate the exact memory locations to a set of potential memory locations. For instance, any memory location accessed by the code in Figure 5.11 is reachable by the procedure in Figure 5.14a. The assertions in Figure 5.14b must hold for any possible memory location returned by the procedure in Figure 5.14a to pass verification.

Figure 5.14b shows the main procedure. In line 3, the verifier assumes both arrays hold arbitrary values, making the verification input-independent. Then, the verifier arbitrarily picks two memory accesses in lines 9-10. Each memory access can capture any memory access during the whole execution of `transformVector`. The assertion describes the dependence constraint to be proved that for any two valid memory accesses (line 11), if they are in different subgraphs (line 12), they must be independent. Lines 13-15 describe the independence, where they either touch different arrays or different indices, or they are both loads. If the assertion always holds, then it is safe to parallelise `loop_0` and `loop_1`.

Our tool flow generates $\frac{k(k-1)}{2}$ assertions for k subgraphs, because that is the number of ways of picking 2 subgraphs from k. The subgraphs are rescheduled based on the verification results. If a subgraph is independent of any its preceding subgraphs within a distance of n, it can simultaneously start with its $(m-n)$th last subgraph. In the case of two or more consecutive subgraphs that are all mutually independent, it is straightforward to schedule them all in parallel. This can be optimised by applying loop fusion passes [165] at the source level to reduce the number of loops while exploiting the parallelism, but our approach also enables hardware parallelism for loops that cannot be fused.

However, a sequence of subgraphs that are neither completely independent nor completely dependent may result in several possible solutions. For instance, the CFG in Figure 5.15a1

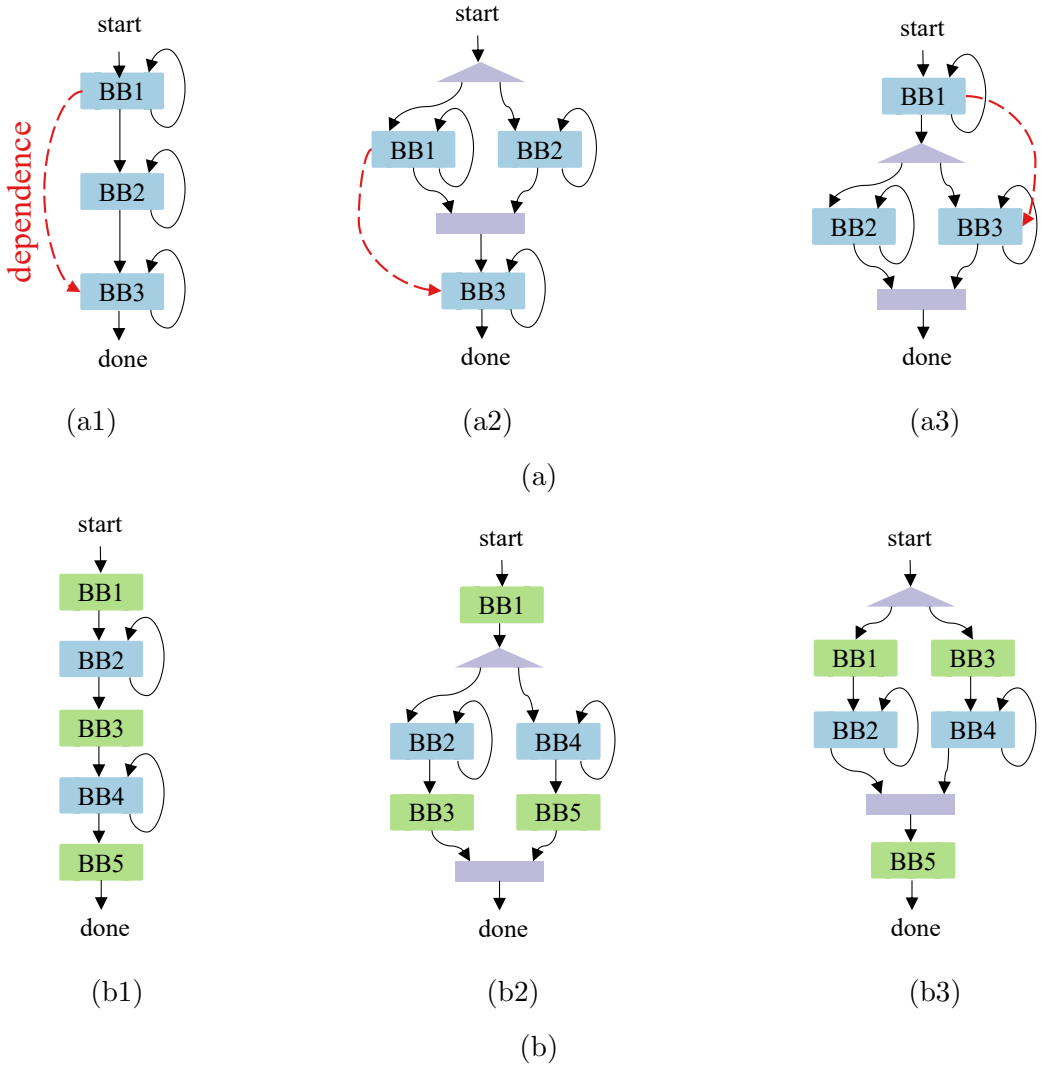


Figure 5.15: A CFG may be parallelised differently, depending on (a) parallelising in top-to-bottom/bottom-to-top order, and (b) grouping BBs with the loop before/after. The dashed arrows represent the memory dependence.

contains three consecutive loops, BB1, BB2, and BB3. BB1 and BB2 can be parallelised, as can BB2 and BB3, but BB1 and BB3 cannot. We therefore have to choose between parallelising them as in Figure 5.15a2 or in Figure 5.15a3. Our current approach greedily parallelises BBs in top-to-bottom order, so yields Figure 5.15a2 by default, but this order can be overridden via a user option. It may be profitable in future work to consider Figure 5.15a3 as an alternative if BB2 and BB3 have more closely matched latencies.

Second, the BBs between sequential loops can be included in a subgraph of either loop, resulting in several solutions. For instance, Figure 5.15b1 can be parallelised to Figure 5.15b2 or to Figure 5.15b3. In Figure 5.15b2, BB2 is grouped with its succeeding loop BB3, and so is BB4. In Figure 5.15b3, the BBs are grouped with their preceding loops. This may result in different verification results which affect whether the subgraphs can be parallelised. For instance, if BB3 depends on BB2, then Figure 5.15b2 is memory-legal and Figure 5.15b3 is invalid (our tool flow will keep the CFG as in Figure 5.15b1). This grouping can be controlled via a user option.

Hardware Transformation

We here explain how to construct dynamically scheduled hardware in which BBs can start simultaneously. First, we illustrate how to insert additional components to enable BB parallelism. Second, we show how to simplify the data flow to avoid unnecessary stalls for subgraphs.

Components Insertion for Parallelism

With given sets of subgraphs that start simultaneously, our tool flow inserts additional components into the dynamically scheduled hardware to enable parallelism. For each set, our tool flow first finds the start of the first subgraph and the exit of the last subgraph in the program order. The trigger of the first subgraph is forked to trigger the other subgraphs in the set. The exit of the last subgraphs is joined with the exits of the other subgraphs and then triggers its succeeding BB. For the example in Figure 5.13b, the start of the function is forked to trigger both `loop_0` and `loop_1`. A join is used to synchronise the BB starting siganls in `loop_0` and

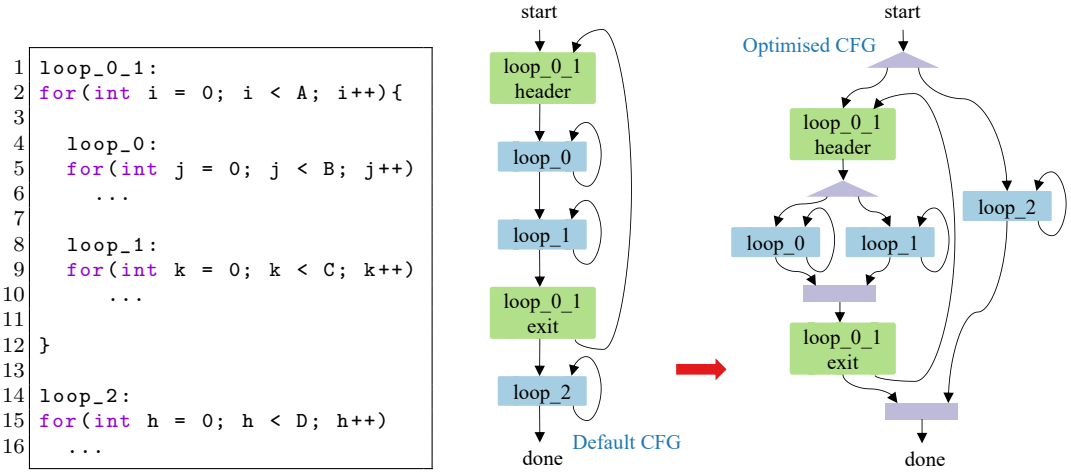
`loop_1`. The join waits for all the BBs in both loops to start and then starts the succeeding BB of the loops.

The BB starting order is now out-of-order, but the computed data cannot be proven independent must be processed in-order. The transformation above ensures the order of data does not affect the correctness. Outside of the loops to be parallelised, the order remains unmodified. When each parallelised loop starts, a token enters the loop and circulates through the loop exactly as the program order. The parallelised loop outputs are synchronised by the join, thus, everything that happens later remains in order. Only the BB orders among these parallelised loops are out-of-order, which have been proven independent.

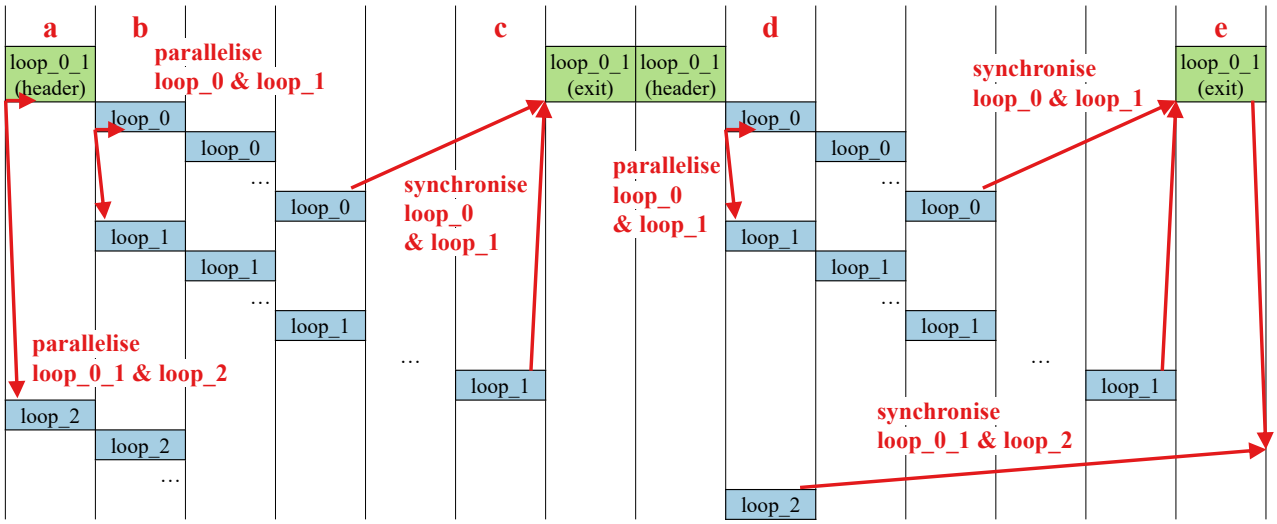
An advantage of such transformation is that the execution of parallelised subgraphs and their succeeding BB are in parallel, although they still start in order. The memory dependences between these subgraphs and the succeeding BB are still respected at run time as they start in order. This effect qualitatively corresponds to what standard dynamically scheduled hardware exhibits yet, in that case, only on a single BB at a time. Compared to traditional static scheduling, which only starts the succeeding BB when all the subgraphs finish execution, our design can achieve better performance.

Figure 5.16 shows an example of parallelising nested parallel subgraphs. The code contains two sequential loops, `loop_0_1` and `loop_2`. Loop `loop_0_1` is a nested loop that contains two sequential loops, `loop_0` and `loop_1`. For simplicity, assume that there is no dependence between any two loops.

Our tool flow constructs two sets of subgraphs in two depths, allowing more parallelism in the CFGs. One set contains `loop_0_1` and `loop_2`, and the other set contains `loop_0` and `loop_1`. The transformation of CFG is illustrated in Figure 5.16b. `loop_0_1` and `loop_2` are parallelised at the start the program, and `loop_0` and `loop_1` are further parallelised inside `loop_0_1`. The corresponding BB starting schedule is demonstrated in Figure 5.16c, which only shows the time when each BB starts. A BB may have a long latency and execute in parallel with other BBs.



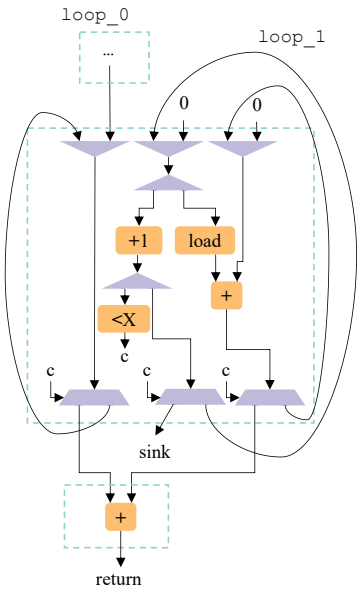
(a) Source (b) CFG transformation.

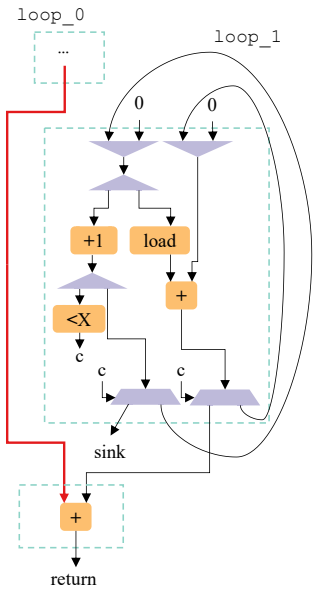


(c) Parallel BB starting schedule.

Figure 5.16: An example of parallelising BB starting schedule by CFG transformation. There are two sets of sequential loops in different depths. Assuming all the loops are independent, each set of sequential loops start simultaneously after the transformation in (b). (c) only shows the time when a BB starts, where a BB may take multiple cycles to execute.

```
1 loop_0:
2 for(j=0; j<Y; j++)
3   t += B[j];
4
5 loop_1:
6 for(i=0; i<X; i++)
7   s += A[i];
8
9 return s+t;
```





(a) Source. (b) Default dataflow. (c) Optimized dataflow.

Figure 5.17: An example of simplifying data flow for live variables. `t` is a live variable in line 3, but not used in `loop_1`. `t` circulates in `loop_1` to preserve liveness but is seen as data dependences, stalling `loop_1` before `t` is valid. Our tool flow identifies and removes these cycles, such that `loop_1` can start earlier.

Forwarding Variables in Data Flow

The second step is to simplify the data flow of live variables for parallelising sequential loops. Dynamatic directly translates the CFDG of an input program into a hardware dataflow graph. In the data flow graph, each vertex represents a hardware operation, and each edge represents a data dependence between two operations.

The data flow of a loop uses cycles for each variable that has carried dependence. The data circulates in the cycle and updates its value in each iteration. However, such an approach also maintains all the live variables in these cycles while executing a loop, even when they are not used inside the loop. The edges of these cycles are seen as data dependences in the hardware, where the edges for unused live variables could cause unnecessary pipeline stalls.

For example, the loops in Figure 5.17a can be parallelised. `loop_0` accumulates array `B` onto `t`, and `loop_1` accumulates array `A` onto `s`. The sum of `s` and `t` is returned. The dataflow graph of `loop_1` is shown in Figure 5.17b. The loop iterator `i` and the variable `s` have carried dependence in `loop_1`. They are kept and updated in the middle and right cycles. The result

of `loop_0`, `t`, is still live and required by addition in line 9. `t` is kept in the left cycle, circulating with `i` and `s`.

`loop_1` is stalled by the absence of `t` even when parallelised with `loop_0`, but `t` is not needed by `loop_1`. In order to remove these unnecessary cycles, our tool flow checks whether a live variable is used in the loop. If it is not, our tool flow removes the corresponding cycle and directly forwards the variable to its next used BB. Figure 5.17c illustrates the transformed dataflow graph. `t` is now directly forwarded to the final adder, enabling two loops to start simultaneously.

LSQ Handling

The parallel BB schedule also affects the LSQs. First, the original Dynamatic starts BB sequentially, where the LSQ expects sequential BB allocation. Our parallelised schedule allows multiple BBs to start simultaneously; therefore, we place a round-robin arbiter for the LSQ to serialise the allocations. The out-of-order allocation still preserves correctness as the simultaneous BB requests have been statically proven independent by our approach in Section 5.2.4.

Second, the arbiter may cause deadlock if the LSQ depth is not sufficient to consume and reorder all memory accesses (*e.g.* a later access may be stuck in an LSQ waiting for a token from an earlier access, but the earlier access cannot enter the LSQ if it is full, thus never supplying the token). This issue has been extensively explored in the context of shared resources in dataflow circuits [166]; similarly to what is suggested in this work, the appropriate LSQ size could be determined based on the number of overlapping loop iterations and their IIs. Although systematically determining the minimal allowed LSQ depth is out of the scope of this work, we here assume a conservative LSQ size that ensures such deadlock never occurs in the benchmarks we consider. We note that minimising the LSQ is orthogonal to our contribution and could only positively impact our results (by reducing circuit area and improving its critical path).

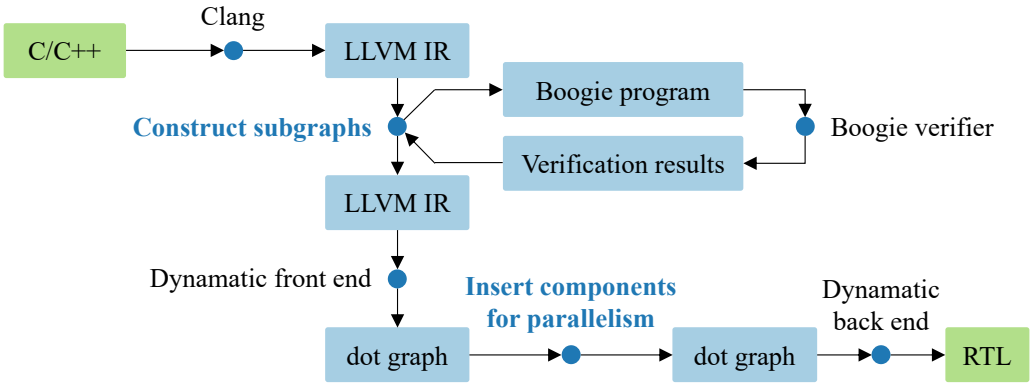


Figure 5.18: Our work integrated into the open source Dynamatic tool [13]. Our contribution is highlighted in bold text.

Tool Flow

Our tool flow is implemented as a set of LLVM passes and integrated into the open-sourced HLS tool Dynamatic for prototyping. As illustrated in Figure 5.18, the input C program is first lowered into LLVM IR and analysed by our subgraph constructor. It generates Boogie assertions and calls the Boogie verifier to automatically verify the absence of dependence between any two subgraphs. Then the constructor constructs sets of subgraphs and reschedules them. The front end of Dynamatic translates the LLVM IR into a dot graph that represents the hardware netlist. Our back-end tool flow inserts additional components and simplifies the unnecessary cycles for the live variables, resulting in a new hardware design in the form of a dot graph. Finally, the back end of Dynamatic translates the dot graph to the final hardware design.

5.2.5 Experiments

We compare our work with Xilinx Vivado HLS [14], the original Dynamatic [13], and Dynamatic with C-slow pipelining [154]. To make the comparison as controlled as possible, all the approaches only use scheduling, pipelining and array partitioning. We use two benchmark sets to evaluate the designs in terms of total circuit area and wall-clock time. Cycle counts were obtained using the Vivado XSIM simulator, and area results were obtained from the post-Place & Synthesis report in Vivado. We used the UltraScale+ family of FPGA devices for experiments, and the version of Xilinx software is 2019.2.

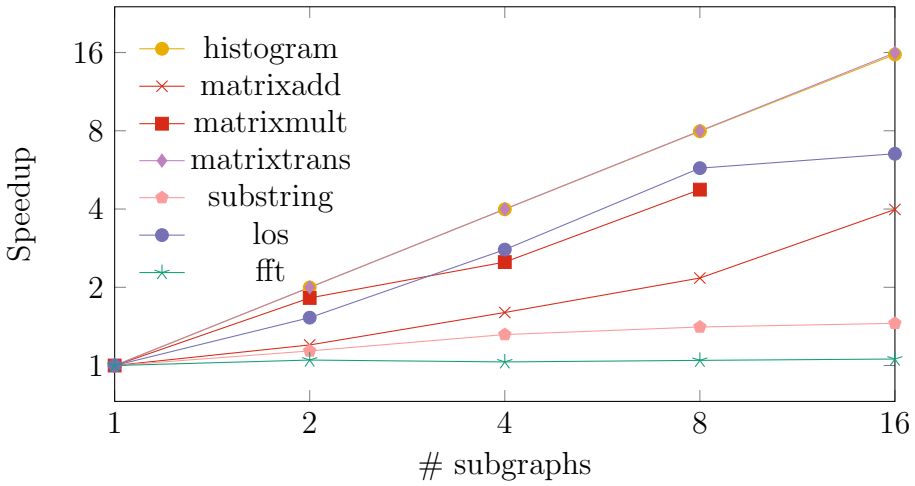


Figure 5.19: Speedup, compared to original Dynamatic, as more subgraphs in the CFG are parallelised.

Results

Figure 5.19 assesses the extent to which more parallelisation of subgraphs leads to more speedups compared to the original Dynamatic, using the seven LegUp benchmarks. We see that all the lines except `matrixadd` indicate speedup factors above 1. Placing more subgraphs in parallel leads to more speedup, with `histogram` and `matrixtrans` achieving optimal speedups. In the `matrixadd` benchmark, two reasons for the lack of speedup are: 1) that there are other parts of the CFG that have to be started sequentially, and 2) that the memory is naively partitioned in a block scheme, so the memory bandwidth is limited and there is serious contention between BBs when accessing the LSQs.

Detailed results for the LegUp benchmarks using eight subgraphs are shown in Table 5.3a. We observe the following:

1. Static scheduling (Vivado HLS) is the clear winner in terms of area (see columns 'LUTs' and 'DSPs'), but in the context of dynamic scheduling, our approach brings only a negligible area overhead because we only insert small components into the hardware.

2. Our approach requires substantially fewer cycles than the original Dynamatic thanks to the parallelism it exploits between BBs (see column 'Cycles').

3. The only benchmark where C-slow pipelining wins is the `matrixmult` benchmark (see

Table 5.3: Evaluation of our work on two benchmark sets. For each benchmark, we highlight the **best** results in each dimension.

(a) Evaluation on the LegUp benchmarks. vhls = Vivado HLS; dhls = original Dynamatic; cslow = C-slow pipelining by Cheng *et al.* [154]; ours = our work; both = our work + C-slow pipelining. The code size lists the number of loops, BBs, instructions and extracted subgraphs.

Benchmarks	Code size				LUTs (1000s)					DSPs					Cycles (1000s)					Fmax (MHz)					Relative area-delay product				
	loops	bbs	insts	graphs	vhls	dhls	ours	cslow	both	vhls	dhls	ours	cslow	both	vhls	dhls	ours	cslow	both	vhls	dhls	ours	cslow	both	vhls	dhls	ours	cslow	both
histogram	9	91	384	9	**1.67**	156	156	156	156	0	0	0	0	0	197	317	**39.8**	317	**39.8**	**464**	57.7	58.3	57.7	58.3	**1**	1212	150.9	1212	150.9
matrixadd	8	17	112	8	**1.11**	9.15	9.17	9.15	9.17	**2**	30	30	30	30	262	106	**48.9**	106	**48.9**	**159**	110	110	110	110	**1**	4.8	2.2	4.8	2.2
matrixmult	72	145	1521	72	**6.87**	79.4	79.8	101	100	**5**	320	320	320	320	4195	1090	229	**164**	**164**	**155**	123	104	83.3	72.3	1	3.8	**0.9**	1.1	1.2
matrixtrans	8	17	81	8	**0.103**	2.67	2.69	2.67	2.69	0	0	0	0	0	65.6	65.6	**8.2**	65.6	**8.2**	**562**	227	210	227	210	**1**	64	8.7	64	8.7
substring	16	54	255	8	**0.938**	14.4	14.6	14.4	14.6	0	0	0	0	0	**98.3**	217	154	217	154	**470**	126	129	126	129	**1**	126	88.9	126	88.9
los	24	89	513	8	**2.68**	46.5	46.1	46.5	46.1	0	0	0	0	0	48.8	114	**19.9**	114	**19.9**	281	272	**282**	272	**282**	**1**	41.8	7	41.8	7
fft	24	65	457	8	**2.65**	351	351	351	351	**16**	192	192	192	192	86	5.39	**5.15**	5.39	**5.15**	**155**	81.8	103	81.8	103	**1**	15.7	12	15.7	12
Norm. median					1	**1**	1	**1**		1	1	1	1		1	**0.21**	1	**0.17**		1	1.01	1	1.01		**1**	41.8	8.7	41.8	8.7

(b) Evaluation on the C-slow pipelining benchmarks. unroll = original Dynamatic taking the program where all outermost loops are unrolled by a factor of 8; ours = unroll + our work; both = unroll + our work + C-slow pipelining.

Benchmarks	Code size (unrolled)				LUTs (1000s)					DSPs					Cycles (1000s)					Fmax (MHz)					Relative area-delay product				
	loops	bbs	insts	graphs	dhls	unroll	ours	cslow	both	dhls	unroll	ours	cslow	both	dhls	unroll	ours	cslow	both	dhls	unroll	ours	cslow	both	dhls	unroll	ours	cslow	both
trVecAccum	16	49	249	8	**18.9**	149	150	19.8	153	**5**	40	40	**5**	40	389	393	161	256	**33.2**	**188**	132	121	157	117	1	11.27	5.05	**0.83**	1.11
covariance	48	113	593	24	27.1	56.6	55.5	**26.8**	65.7	**9**	72	72	**9**	72	698	605	77.1	263	**33.6**	72.1	**132**	102	89.7	102	1	0.99	0.16	0.30	**0.08**
syr2k	24	49	385	8	**4.14**	30.5	30.7	4.57	34.4	**19**	152	152	**19**	152	674	602	84.1	255	**33.9**	125	126	98.9	**130**	128	1	6.53	1.16	**0.40**	0.41
gesummv	16	33	297	8	**3.96**	26.9	26.9	4.56	30.2	**18**	144	144	**18**	144	797	787	327	524	**66.6**	128	162	**163**	119	120	1	5.33	2.20	0.82	**0.68**
Norm. median					1	7.10	**7.09**	1.08	7.85	1	8	8	1	8	1	1	**0.27**	0.52	**0.07**	1	1.13	1.03	0.99	0.98	1	5.93	1.68	0.61	**0.54**

column 'Cycles'), which contains nested loops. Fortunately, our approach is compatible and complementary to C-slow pipelining, so we can apply both techniques simultaneously to reach the best of both worlds (see the 'both' columns).

4. Our approach achieves 1.05-9.05× speedup in total cycles. We further observe that the area-delay products we obtain are significantly smaller than those of the original Dynamatic.

5. Although Vivado HLS has low performance in cycles, its high clock frequency makes it win for `histogram` and `substring`. Also, it uses if-conversion to simplify BBs (unlike our work), which results in fewer BBs. The BBs in the innermost subgraphs still start sequentially, leading to large cycle counts.

For the C-slow pipelining benchmark set (Table 5.3b), we make the following observations:

1. The area-delay product of Dynamatic is significantly worse than Vivado HLS because the version of Dynamatic we used does not support resource sharing leading to significant area overhead (although it is now supported [166]).

2. Unrolling alone is not enough to obtain substantial speedups because the BBs still have to start sequentially (see column 'Cycles → unroll').

3. C-slow pipelining enables 1.53-6.65× speedup with only up to 15% area overhead. Our approach achieves 1.5-7.85× speedup for the unrolled benchmarks. By applying both techniques simultaneously on the unrolled programs, we achieve 1.05-20.6× speedup with up to 16% area overhead. That significant speedup can be attributed in part to the reordering of BBs.

5.2.6 Summary

Existing dynamically scheduled HLS tools require all BBs to start in strict program order, in order to respect any inter-BB dependences, regardless of whether dependences are actually

present. This leads to missed opportunities for performance improvements by having BBs start simultaneously.

We propose an automated approach to lifting this restriction. We show how to statically identify sequences of consecutive subgraphs in the CFG of a program and reschedule them (with the help of the Microsoft Boogie verifier) to start simultaneously. We then show how to map the optimised schedule into efficient hardware designs. The performance gain is significant (and can be further improved with C-slow pipelining), while the area overhead is negligible. Our plan for future work is to automate the process of optimising subgraph configurations for arbitrary programs in this framework.

Chapter 6

Conclusion

This thesis extends the state-of-the-art of scheduling techniques in high-level synthesis. We tackled the problems in existing scheduling formulation and proposed an efficient approach that combines the best of two worlds, static scheduling and dynamic scheduling. The technical contributions of this thesis are generally in four aspects as follows.

Combining Static and Dynamic Scheduling

The existing formulation only supports modelling one of the existing approaches. Users have to manually choose one of them and compromise area or performance when designing their custom hardware. Sometimes this cost could be significant. In Chapter 3, we proposed a novel formulation that combines both static scheduling and dynamic scheduling for HLS. Our formulation identified the gap between the theoretical dependence constraints and the implemented dependence constraints. This guided us to find optimisation opportunities towards optimal scheduling. We showed how to use the proposed formulation for implementing an efficient scheduling approach with better hardware performance, and more hardware optimisations for better hardware performance by relaxing the conservative restrictions from the existing formulations in our model.

Based on the formulation introduced in Chapter 3, we implemented an HLS tool named DASS

with an optimised scheduling approach from existing static scheduling and dynamic scheduling. DASS allows part of the hardware to be statically scheduled in the form of static islands and integrated into a dynamically scheduled hardware design. We proposed an efficient hardware design that enables hardware components with different scheduling approaches to communicate correctly with high performance. Our experimental results showed that DASS can achieve the best performance among the state-of-the-art HLS tools for FPGAs. The area overhead was between static scheduling and dynamic scheduling as expected.

Finding and Finessing Static Islands

A problem with DASS was that the implementation in Chapter 3 needs to be manually configured by experts to produce efficient hardware designs. In Chapter 4, we proposed a few static analysis techniques that automate these steps. First, we formalised the code features that are amenable for static scheduling both theoretically and practically. We then implemented heuristic-driven approaches in a few passes by applying the practical constraints for determining a good set of static islands from an arbitrary program, which used to be done by experts. Our experimental results have shown that our automatically partitioned solutions are as good as the manually optimised designs in both performance and area.

Second, the static islands required a set of scheduling constraints to be efficiently scheduled. Typically, hardware designers needed to investigate the throughputs of two connected components and balance their throughputs for efficient resource utilisation. Again, this required expert-level knowledge. The case of analysing dynamically scheduled hardware behaviour was more challenging as the exact behaviour is unknown at compile time. We proposed a probabilistic approach to estimate the throughput of dynamically scheduled hardware statically by modelling them in Petri nets. The tool then uses the estimated throughput for balancing the throughputs of static islands. Our approach has shown significant area saving when compared with existing throughput analysis approaches, and significant speedup in compilation time when compared with exhaustive search.

Parallelisng Control Flow

Chapter 3 formalised existing static scheduling and dynamic scheduling in a unified model. Chapter 4 used the model to optimise the statically scheduled hardware. However, the dynamically scheduled hardware still has performance space for performance improvements. Chapter 5 presented two techniques that optimise existing dynamic scheduling at the control flow level for better performance. First, we re-visited the concept of C-slow pipelining and achieves it in dynamically scheduled HLS hardware. We adopted the formulation of dynamic C-slow pipelining in the proposed scheduling model and propose an efficient hardware implementation for C-slowing nested loops. Second, we proposed an approach to statically check the dependence between sequential loops and efficiently parallelise them in hardware. Our experimental results have shown that both approaches significantly improve performance with a negligible area overhead.

6.1 Outlook

The research outcome in this thesis has contributed to the open-sourced HLS community. The concept of DASS has been adapted to an open-sourced HLS tool named CIRCT, implemented in MLIR. The existing MLIR framework contains a few dialects that are used in existing MLIR-based HLS tools [167]:

affine: The `affine` dialect represents affine operations, such as affine loops and affine memory operations.

scf: The `scf` dialect represents structured control flow operations, such as `for` loops, `while` loops, `do while` loops and `if` statements.

cf: The `cf` dialect represents unstructured control flow operations, such as general branch operations.

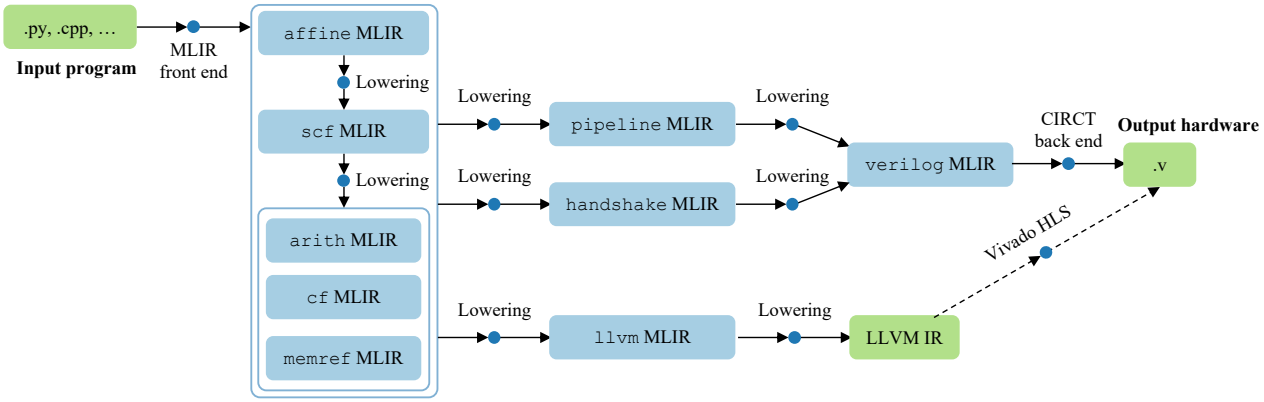


Figure 6.1: MLIR lowering process in CIRCT HLS [2].

arith: The `arith` dialect represents basic arithmetic and logical operations for scalars and vectors.

memref: The `memref` dialect represents general memory operations.

pipeline: The `pipeline` dialect represents the timing information of an HLS program after hardware parallelism.

handshake: The `handshake` dialect represents a program in a netlist of operations connected through handshaking signals.

verilog: The `verilog` dialect represents a program in a low-level hardware abstraction that can be directly translated into Verilog code.

Most MLIR-based HLS tools use the dialects above as part of their intermediate representations. For instance, Figure 6.1 shows the tool flow of an HLS tool implemented in MLIR, named CIRCT HLS [2].

1. CIRCT accepts software programs using MLIR front end. For instance, C programs are parsed by a front-end tool named Polygeist [168] and Python programs are parsed by a front-end tool named Torch-MLIR [169].

2. Then the program is first lowered to the highest IR. Each operation in the source is mapped to the highest possible dialect. For instance, affine operations are represented in `affine` dialect, and `for` loops are represented in `scf` dialect. General operations are

represented in relatively low-level dialects. For instance, non-affine loads and stores are represented in `memref` dialect.

3. Then the program is lowered progressively from each dialect. For instance, an affine `for` loop is firstly lowered to `affine` dialect for affine optimisation, such as loop unrolling. It is then lowered to `scf` dialect for control flow optimisation, such as loop rotation which transforms a `for` loop to a `do...while` loop. Next, it is further lowered to a mixture of `arith`, `cf` and `memref` dialects for instruction-level optimisation, such as constant propagation. These steps exist in the software design flow and can be directly reused for HLS flows.

4. Next, the IR can be lowered to the `pipeline` dialect for static scheduling, which adds timing constraints to each operation, or the `handshake` dialect for dynamic scheduling, which add spatial constraints to each operation. These two dialects can be mixed in a program. This concept is inspired by the DASS implementation in Chapter 3 but there is no implementation in CIRCT.

5. The lowered program is then lowered to `verilog` dialect and emitted in Verilog as the output hardware design.

There is also a back end that emits LLVM IR by lowering the software IR to the LLVM dialect and produces RTL designs through the Vitis HLS open-sourced LLVM front end [76]. This approach keeps pure software representation in the whole MLIR flow but has less granularity control compared to lowering through `verilog` dialect. With these multi-level abstractions, tool designers can insert HLS-specific passes into the most amenable abstractions in the form of MLIR dialects, enabling producing efficient hardware [2, 76].

The research works in this thesis can be directly transferred to other HLS tools such as CIRCT with engineering efforts, closing the gap of design quality between HLS-generated hardware and manually implemented RTL designs. Some future works of this research are as follows:

Beyond Synthesis

Existing HLS tools generate high-level RTL descriptions in the form of finite-state machines. Then the vendor tools in the back end perform logic synthesis to map the hardware design onto a specific architecture. The output of the logic synthesis tool is a netlist, which contains a set of components in the logic cells of a specific target. A problem with this implementation is that the information at high level might be lost in the logic synthesis process. This might miss opportunities for further optimisation. A possible direction is to include the implementation process in the HLS flow, such as AutoBridge [170] and RapidStream [171], where an HLS tool might guide the implementation tool for a better hardware design.

Beyond FPGA

Existing HLS tools target ASIC or FPGAs. Recently, there has been interest to extend HLS tools to support more reconfigurable devices, such as Coarse-Grained Reconfigurable Architectures (CGRAs). Compared with an FPGA, a CGRA contains larger unit cells, named functional units. A CGRA has reconfigurable between ASIC and FPGAs. However, efficiently mapping programs onto a CGRA is still challenging. A possible direction is to extend the scheduling formulation to support CGRAs by adding CGRA-specific intrinsics.

Beyond C/C++ Input

Most HLS tools accept C/C++ programs as they are widely used by software engineers. However, C/C++ languages are relatively low-level when compared to languages such as Python [40], Julia [42] and Coq [172]. Extending HLS tools to support different input languages may help the static analysis in the compilation flow by utilising the features of these input languages, such as using PyTorch [173] for generating efficient hardware designs for machine learning applications.

The future computation platforms are going to be multi-targeted for better energy efficiency, where each custom hardware computes for its amenable applications. These targets are virtu-

alised, minimising the amount of domain knowledge required from experts, hence these technologies can be applied at scale and be accessible to a broader range of people. My long-term ambition is to build a tool flow that automatically schedules and maps applications onto a multi-target system, pushing the automation on the optimisation for both software and hardware. I believe that this thesis could be a stepping stone for achieving this goal in the next few decades, which makes HLS more accessible to a broader range of people.

Real-World Applications

The benchmarks evaluated in this thesis are relatively small but contain code patterns that can be found in various real-world applications. For instance, it is common to process images with irregular shapes in computer vision, such as 360-Degree Video Rendering [113] and biophotonic cancer tumour detection [98]. The program behaviour of these applications is often input-dependent. Dynamic scheduling can efficiently balance the workload with the handshake interface between hardware operations, leading to potentially high throughput.

The control flow of network switches for edge computing can also be input-dependent and complex. For instance, it is essential for the network switches to compute in parallel with high performance [174]. Dynamic scheduling can significantly improve the hardware performance for unpredictable inputs.

Efficient acceleration of sparse matrix computation is also an open challenge. Existing approaches encode the sparse matrices into dense matrices for efficient acceleration in a statically scheduled pipeline [175, 176]. However, the encoding process could cause performance overhead, where dynamic scheduling can directly handle sparse matrices at run time and potentially achieves high performance.

6.2 Summary

In past decades, many efforts have been made in the HLS area to improve hardware performance, including memory partitioning [103], loop optimisation [159] and scheduling [46]. In this thesis, we propose the DASS HLS tool that combines the best of two worlds in scheduling. We push the performance limits of HLS hardware designs using efficient dynamic and static scheduling. We believe DASS can close the gap in the hardware design quality between HLS designs and manually-optimised designs.

Bibliography

[1] J. Cheng, L. Josipović, G. A. Constantinides, P. Ienne, and J. Wickerson, "DASS: Combining Dynamic and Static Scheduling in High-level Synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 3, pp. 628–641, 2022.

[2] C. contributors, "CIRCT: Circuit IR Compilers and Tools," https://github.com/llvm/circt/tree/main/, 2021.

[3] "Energy consumption of ICT," 2022. [Online]. Available: https://post.parliament.uk/research-briefings/post-pn-0677/

[4] A. Nohl, F. Schirrmeister, and D. Taussig, "Application specific processor design: Architectures, design methods and tools," in *2010 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2010, pp. 349–352.

[5] K. Neshatpour, H. M. Mokrani, A. Sasan, H. Ghasemzadeh, S. Rafatirad, and H. Homayoun, "Architectural Considerations for FPGA Acceleration of Machine Learning Applications in MapReduce," in *Proceedings of the 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, ser. SAMOS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 89–96. [Online]. Available: https://doi.org/10.1145/3229631.3229639

[6] N. Ratha and A. Jain, "FPGA-based computing in computer vision," in *Proceedings Fourth IEEE International Workshop on Computer Architecture for Machine Perception. CAMP'97*, 1997, pp. 128–137.

[7] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An Introduction to High-Level Synthesis," *IEEE Design Test of Computers*, vol. 26, no. 4, pp. 8–17, July 2009.

[8] "Reprogrammable HPC," 2022. [Online]. Available: https://www.scientific-computing.com/feature/reprogrammable-hpc

[9] "Microsoft Project Catapult," 2022. [Online]. Available: https://www.microsoft.com/en-us/research/project/project-catapult/

[10] "Amazon EC2 F1 Instances," 2022. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/

[11] D. Thomas and P. Moorby, *The Verilog hardware description language*. Springer Science & Business Media, 2008.

[12] V. G. Castellana, A. Tumeo, and F. Ferrandi, "High-level synthesis of memory bound and irregular parallel applications with Bambu," in *2014 IEEE Hot Chips 26 Symposium (HCS)*. Cupertino, CA: IEEE, Aug 2014, pp. 1–1.

[13] L. Josipović, R. Ghosal, and P. Ienne, "Dynamically Scheduled High-level Synthesis," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18. Monterey, CA: ACM, 2018, pp. 127–136.

[14] Xilinx Vivado HLS, 2022. [Online]. Available: https://www.xilinx.com/support/documentation-navigation/design-hubs/dh0012-vivado-high-level-synthesis-hub.html

[15] Intel HLS Compiler, 2022. [Online]. Available: https://www.intel.co.uk/content/www/uk/en/software/programmable/quartus-prime/hls-compiler.html

[16] Stratus High-Level Synthesis, 2021. [Online]. Available: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html

[17] Catapult High-Level Synthesis, 2021. [Online]. Available: https://www.mentor.com/hls-lp/catapult-high-level-synthesis

[18] S. D. Brown, R. J. Francis, J. Rose, and Z. G. Vranesic, *Field-programmable gate arrays*. Springer Science & Business Media, 1992, vol. 180.

[19] U. Meyer-Baese and U. Meyer-Baese, *Digital signal processing with field programmable gate arrays*. Springer, 2007, vol. 65.

[20] J. Lamoureux and W. Luk, "An overview of low-power techniques for field-programmable gate arrays," in *2008 NASA/ESA Conference on Adaptive Hardware and Systems*. IEEE, 2008, pp. 338–345.

[21] J. Cong and Y. Ding, "On area/depth trade-off in LUT-based FPGA technology mapping," in *Proceedings of the 30th International Design Automation Conference*, 1993, pp. 213–218.

[22] G. Conde and G. W. Donohoe, "Reconfigurable Block Floating Point Processing Elements in Virtex Platforms," in *2011 International Conference on Reconfigurable Computing and FPGAs*, 2011, pp. 509–512.

[23] B. Ronak and S. A. Fahmy, "Mapping for maximum performance on FPGA DSP blocks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 4, pp. 573–585, 2015.

[24] P. Alfke, "Creative uses of block RAM," *White Paper: Virtex and Spartan FPGA Families, Xilinx*, 2008.

[25] Z. Navabi, *VHDL: Analysis and modeling of digital systems*. McGraw-Hill, Inc., 1997.

[26] R. L. Rudell, *Logic synthesis for VLSI design*. University of California, Berkeley, 1989.

[27] G. D. Hachtel and F. Somenzi, *Logic synthesis and verification algorithms*. Springer Science & Business Media, 2007.

[28] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *International Workshop on Field Programmable Logic and Applications*. Springer, 1997, pp. 213–222.

[29] K. D. Pham, E. Horta, and D. Koch, "BITMAN: A tool and API for FPGA bitstream manipulations," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 894–897.

[30] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[31] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. Monterey, CA, USA: ACM, 2011, pp. 33–36.

[32] C. Lattner, "LLVM and Clang: Next generation compiler technology," in *The BSD conference*, vol. 5, 2008, pp. 1–20.

[33] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software implemented fault tolerance," in *International symposium on Code generation and optimization*. IEEE, 2005, pp. 243–254.

[34] Intel Compiler, 2022. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler.html#gs.sa60u7

[35] H. Ye, C. Hao, J. Cheng, H. Jeong, J. Huang, S. Neuendorffer, and D. Chen, "ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022.

[36] Y.-H. Lai, Y. Chi, Y. Hu, J. Wang, C. H. Yu, Y. Zhou, J. Cong, and Z. Zhang, "HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019, pp. 242–251.

[37] H. Folmer, R. d. Groote, and M. Bekooij, "High-Level Synthesis of Digital Circuits from Template Haskell and SDF-AP," in *International Conference on Embedded Computer Systems*. Springer, 2022, pp. 3–27.

[38] C. Baaij, "Cλash: From Haskell to hardware," Master's thesis, University of Twente, 2009.

[39] S. Huang, K. Wu, H. Jeong, C. Wang, D. Chen, and W.-M. Hwu, "PyLog: An Algorithm-Centric Python-Based FPGA Programming and Synthesis Flow," *IEEE Transactions on Computers*, vol. 70, no. 12, pp. 2015–2028, 2021.

[40] G. vanRossum, "Python reference manual," *Department of Computer Science [CS]*, no. R 9525, 1995.

[41] B. Biggs, I. McInerney, E. C. Kerrigan, and G. A. Constantinides, "High-level Synthesis using the Julia Language," in *2nd Workshop on Languages, Tools, and Techniques for Accelerator Design*, ser. LATTE '21, 2022. [Online]. Available: https://capra.cs.cornell.edu/latte22/paper/7.pdf

[42] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM review*, vol. 59, no. 1, pp. 65–98, 2017.

[43] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 519–530. [Online]. Available: https://doi.org/10.1145/2491956.2462176

[44] S. I. Venieris and C.-S. Bouganis, "fpgaConvNet: A framework for mapping convolutional neural networks on FPGAs," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2016, pp. 40–47.

[45] ONNX, 2022. [Online]. Available: https://onnx.ai

[46] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on SDC formulation," in *2006 43rd ACM/IEEE Design Automation Conference*. San Francisco, CA: IEEE, 2006, pp. 433–438.

[47] Z. Zhang and B. Liu, "SDC-based modulo scheduling for pipeline synthesis," in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2013, pp. 211–218.

[48] K. van Berkel, *Handshake Circuits: An Asynchronous Architecture for VLSI Programming*. USA: Cambridge University Press, 1993.

[49] *Syntax-directed Translation of Concurrent Programs into Self-timed Circuits*, 1988, pp. 35–50.

[50] D. Edwards and A. Bardsley, "Balsa: An asynchronous hardware synthesis language," *The Computer Journal*, vol. 45, no. 1, pp. 12–18, 2002.

[51] T. Yoneda, A. Matsumoto, M. Kato, and C. Myers, "High level synthesis of timed asynchronous circuits," in *11th IEEE International Symposium on Asynchronous Circuits and Systems*, 2005, pp. 178–189.

[52] S. Nielsen, J. Sparso, and J. Madsen, "Towards behavioral synthesis of asynchronous circuits - an implementation template targeting syntax directed compilation," in *Euromicro Symposium on Digital System Design, 2004. DSD 2004.*, 2004, pp. 298–305.

[53] S. F. Nielsen, J. Sparsø, J. B. Jensen, and J. S. R. Nielsen, "A behavioral synthesis frontend to the haste/tide design flow," in *2009 15th IEEE Symposium on Asynchronous Circuits and Systems*, 2009, pp. 185–194.

[54] G. Venkataramani, M. Budiu, T. Chelcea, and S. C. Goldstein, "C to Asynchronous Dataflow Circuits: An End-to-End Toolflow," in *IEEE 13th International Workshop on Logic Synthesis (IWLS)*. Temecula, CA: IEEE, Jun 2004.

[55] M. Budiu and S. C. Goldstein, "Pegasus: An Efficient Intermediate Representation," Carnegie Mellon University, Tech. Rep. CMU-CS-02-107, May 2002.

[56] R. Li, L. Berkley, Y. Yang, and R. Manohar, "Fluid: An asynchronous high-level synthesis tool for complex program structures," in *2021 27th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2021, pp. 1–8.

[57] Ian Page and Wayne Luk, "Compiling occam into Field-Programmable Gate Arrays," in *FPGAs, W. Moore and W. Luk, Eds., Abingdon EE&CS Books*, 1991.

[58] Celoxica, "Handel-C," 2005. [Online]. Available: http://www.celoxica.com

[59] L. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 9, pp. 1059–1076, 2001.

[60] L. Josipovic, P. Brisk, and P. Ienne, "An Out-of-Order Load-Store Queue for Spatial Computing," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 5s, sep 2017. [Online]. Available: https://doi.org/10.1145/3126525

[61] L. Josipović, A. Bhattacharyya, A. Guerrieri, and P. Ienne, "Shrink It or Shed It! Minimize the Use of LSQs in Dataflow Designs," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, 2019, pp. 197–205.

[62] M. Alle, A. Morvan, and S. Derrien, "Runtime dependency analysis for loop pipelining in High-Level Synthesis," in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*. Austin, TX: IEEE, May 2013, pp. 51:1–51:10.

[63] J. Liu, S. Bayliss, and G. A. Constantinides, "Offline Synthesis of Online Dependence Testing: Parametric Loop Pipelining for HLS," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. Vancouver, BC: IEEE, May 2015, pp. 159–162.

[64] M. Tan, G. Liu, R. Zhao, S. Dai, and Z. Zhang, "ElasticFlow: A complexity-effective approach for pipelining irregular loop nests," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. Austin, TX: IEEE, Nov 2015, pp. 78–85.

[65] S. Dai, M. Tan, K. Hao, and Z. Zhang, "Flushing-enabled loop pipelining for high-level synthesis," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. San Francisco, CA: IEEE, June 2014, pp. 1–6.

[66] S. Dai, R. Zhao, G. Liu, S. Srinath, U. Gupta, C. Batten, and Z. Zhang, "Dynamic Hazard Resolution for Pipelining Irregular Loops in High-Level Synthesis," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. Monterey, CA: ACM, 2017, pp. 189–194.

[67] L. P. Carloni, "From Latency-Insensitive Design to Communication-Based System-Level Design," *Proceedings of the IEEE*, vol. 103, no. 11, pp. 2133–2151, Nov 2015.

[68] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.* IEEE, 2004, pp. 7–16.

[69] M. Labbé, G. Laporte, I. R. Martín, and J. J. S. González, "The ring star problem: Polyhedral analysis and exact algorithm," *Networks: An International Journal*, vol. 43, no. 3, pp. 177–189, 2004.

[70] A. Darte, R. Schreiber, and G. Villard, "Lattice-based memory allocation," *IEEE Transactions on Computers*, vol. 54, no. 10, pp. 1242–1257, 2005.

[71] Q. Liu, G. A. Constantinides, K. Masselos, and P. Y. K. Cheung, "Automatic On-chip Memory Minimization for Data Reuse," in *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*. Napa, CA, USA: IEEE, April 2007, pp. 251–260.

[72] Y. Wang, P. Li, and J. Cong, "Theory and Algorithm for Generalized Memory Partitioning in High-level Synthesis," in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, ser. FPGA '14. Monterey, California, USA: ACM, 2014, pp. 199–208.

[73] J. Escobedo and M. Lin, "Graph-Theoretically Optimal Memory Banking for Stencil-Based Computing Kernels," in *Proceedings of the 2018 ACM/SIGDA International Sym-*

posium on Field-Programmable Gate Arrays, ser. FPGA '18. Monterey, CALIFORNIA, USA: ACM, 2018, pp. 199–208.

[74] W. Zuo, Y. Liang, P. Li, K. Rupnow, D. Chen, and J. Cong, "Improving High Level Synthesis Optimization Opportunity through Polyhedral Transformations," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 9–18. [Online]. Available: https://doi.org/10.1145/2435264.2435271

[75] J. Wang, L. Guo, and J. Cong, "AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 93–104. [Online]. Available: https://doi.org/10.1145/3431920.3439292

[76] R. Zhao, J. Cheng, W. Luk, and G. A. Constantinides, "POLSCA: Polyhedral High-Level Synthesis with Compiler Transformations." in *2022 32th International Conference on Field-Programmable Logic and Applications (FPL)*, 2022.

[77] G. Mével and J.-H. Jourdan, "Formal Verification of a Concurrent Bounded Queue in a Weak Memory Model," *Proc. ACM Program. Lang.*, vol. 5, no. ICFP, aug 2021. [Online]. Available: https://doi.org/10.1145/3473571

[78] Y. Herklotz, J. D. Pollard, N. Ramanathan, and J. Wickerson, "Formal Verification of High-Level Synthesis," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021. [Online]. Available: https://doi.org/10.1145/3485494

[79] D. Gao and T. Melham, "End-to-End Formal Verification of a RISC-V Processor Extended with Capability Pointers," in *2021 Formal Methods in Computer Aided Design (FMCAD)*, 2021, pp. 24–33.

[80] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[81] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, "cvc5: A Versatile and Industrial-Strength SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, ser. Lecture Notes in Computer Science, D. Fisman and G. Rosu, Eds., vol. 13243. Springer, 2022, pp. 415–442. [Online]. Available: https://doi.org/10.1007/978-3-030-99524-9_24

[82] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *International conference on logic for programming artificial intelligence and reasoning*. Springer, 2010, pp. 348–370.

[83] ——, "This is Boogie 2," June 2008. [Online]. Available: https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/

[84] Cadence, "Jasper RTL Apps," 2022. [Online]. Available: https://www.cadence.com/ko_KR/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html

[85] M. Carter, S. He, J. Whitaker, Z. Rakamarić, and M. Emmi, "SMACK Software Verification Toolchain," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16. Austin, Texas: ACM, 2016, pp. 589–592.

[86] K. R. M. Leino and C. Pit-Claudel, "Trigger Selection Strategies to Stabilize Program Verifiers," in *Computer Aided Verification*, S. Chaudhuri and A. Farzan, Eds. Cham: Springer International Publishing, 2016, pp. 361–381.

[87] Y. Zhou, K. M. Al-Hawaj, and Z. Zhang, "A New Approach to Automatic Memory Banking Using Trace-Based Address Mining," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New

York, NY, USA: Association for Computing Machinery, 2017, p. 179–188. [Online]. Available: https://doi.org/10.1145/3020078.3021734

[88] J. Cheng, S. T. Fleming, Y. T. Chen, J. Anderson, J. Wickerson, and G. A. Constantinides, "Efficient Memory Arbitration in High-Level Synthesis From Multi-Threaded Code," *IEEE Transactions on Computers*, vol. 71, no. 4, pp. 933–946, 2022.

[89] B. Franke, M. O'Boyle, J. Thomson, and G. Fursin, "Probabilistic source-level optimisation of embedded programs," in *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 2005, pp. 78–86.

[90] C. Tse, K. Wang, C. Chung, and K. Tsang, "Parameter optimisation of robust power system stabilisers by probabilistic approach," *IEE Proceedings-Generation, Transmission and Distribution*, vol. 147, no. 2, pp. 69–75, 2000.

[91] L.-N. Pouchet, "Polybench: The polyhedral benchmark suite," *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, vol. 437, 2012.

[92] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii, "CHStone: A benchmark program suite for practical C-based high-level synthesis," in *2008 IEEE International Symposium on Circuits and Systems*, 2008, pp. 1192–1195.

[93] F. H. McMahon, "The Livermore Fortran kernels: A computer test of the numerical performance range," Lawrence Livermore National Lab., CA (USA), Tech. Rep., Dec. 1986.

[94] Y. Zhou, U. Gupta, S. Dai, R. Zhao, N. Srivastava, H. Jin, J. Featherston, Y.-H. Lai, G. Liu, G. A. Velasquez *et al.*, "Rosetta: A realistic high-level synthesis benchmark suite for software programmable fpgas," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018, pp. 269–278.

[95] J. Cheng, "HLS-benchmarks," 2019. [Online]. Available: https://doi.org/10.5281/zenodo.3561115

[96] J. Duprat and J.-M. Muller, "The CORDIC Algorithm: New Results for Fast VLSI Implementation," *IEEE Trans. Comput.*, vol. 42, no. 2, pp. 168–178, Feb. 1993.

[97] E. Wang, J. J. Davis, P. Y. K. Cheung, and G. A. Constantinides, "LUTNet: Rethinking Inference in FPGA Soft Logic," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. San Diego, CA: IEEE, 2019, pp. 26–34.

[98] T. Young-Schultz, L. Lilge, S. Brown, and V. Betz, "Using OpenCL to Enable Software-like Development of an FPGA-Accelerated Biophotonic Cancer Treatment Simulator," in *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 86–96. [Online]. Available: https://doi.org/10.1145/3373087.3375300

[99] levenberg-maquardt-example, 2021. [Online]. Available: https://github.com/leechwort/levenberg-maquardt-example

[100] gram-schmidt, 2021. [Online]. Available: https://github.com/chrundle/gram-schmidt

[101] LosAlamosChessEngine, 2020. [Online]. Available: https://github.com/gfmcknight/LosAlamosChessEngine

[102] Libchaos, 2020. [Online]. Available: https://github.com/maciejczyzewski/libchaos

[103] Y. T. Chen and J. H. Anderson, "Automated generation of banked memory architectures in the high-level synthesis of multi-threaded software," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–8.

[104] B. Barney, "POSIX Threads Programming," 2021. [Online]. Available: https://computing.llnl.gov/tutorials/pthreads

[105] B. R. Rau, "Iterative modulo Scheduling: An Algorithm for Software Pipelining Loops," in *Proceedings of the 27th Annual International Symposium on Microarchitecture*, ser. MICRO 27. New York, NY, USA: Association for Computing Machinery, 1994, p. 63–74. [Online]. Available: https://doi.org/10.1145/192724.192731

[106] A. Canis, S. D. Brown, and J. H. Anderson, "Modulo SDC scheduling with recurrence minimization in high-level synthesis," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, 2014, pp. 1–8.

[107] J. Cheng, J. Wickerson, and G. A. Constantinides, "Exploiting the Correlation between Dependence Distance and Latency in Loop Pipelining for HLS," in *2021 31th International Conference on Field-Programmable Logic and Applications (FPL)*, 2021, pp. 341–346.

[108] L. Josipović, A. Guerrieri, and P. Ienne, "From C/C++ Code to High-Performance Dataflow Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2021.

[109] Charles Seitz, *System Timing*. Addison-Wesley, 1980.

[110] L. Josipović, S. Sheikhha, A. Guerrieri, P. Ienne, and J. Cortadella, "Buffer Placement and Sizing for High-Performance Dataflow Circuits," in *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '20. Monterey, CA: Association for Computing Machinery, 2020, p. 186–196. [Online]. Available: https://doi.org/10.1145/3373087.3375314

[111] S. Stuijk, M. Geilen, B. Theelen, and T. Basten, "Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications," in *2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*. Samos, Greece: IEEE, July 2011, pp. 404–411.

[112] A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, B. D. Theelen, M. R. Mousavi, A. J. M. Moonen, and M. J. G. Bekooij, "Throughput Analysis of Synchronous Data Flow Graphs," in *Sixth International Conference on Application of Concurrency to System Design (ACSD'06)*. Turku, Finland: IEEE, June 2006, pp. 25–36.

[113] Q. Sun, A. Taherin, Y. Siatitse, and Y. Zhu, "Energy-Efficient 360-Degree Video Rendering on FPGA via Algorithm-Architecture Co-Design," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA

'20. New York, NY, USA: Association for Computing Machinery, 2020, p. 97–103. [Online]. Available: https://doi.org/10.1145/3373087.3375317

[114] R. Zhao, H.-C. Ng, W. Luk, and X. Niu, "Towards Efficient Convolutional Neural Network for Domain-Specific Applications on FPGA," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, 2018, pp. 147–1477.

[115] J. de Fine Licht, G. Kwasniewski, and T. Hoefler, "Flexible Communication Avoiding Matrix Multiplication on FPGA with High-Level Synthesis," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 244–254. [Online]. Available: https://doi.org/10.1145/3373087.3375296

[116] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali, "Lonestar: A suite of parallel irregular programs," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, 2009, pp. 65–76.

[117] N. M. Smart and S. Barnett, "Bilinear transformation of multivariable polynomials using the Horner method," *International Journal of Control*, vol. 37, no. 4, pp. 861–865, 1983.

[118] E. Horowitz, "On the substitution of polynomial forms," in *Proceedings of the ACM annual conference*, 1973, pp. 153–158.

[119] M. Ishikawa and G. De Micheli, "A module selection algorithm for high-level synthesis," in *1991., IEEE International Sympoisum on Circuits and Systems*, 1991, pp. 1777–1780 vol.3.

[120] I. Ahmad, M. K. Dhodhi, and C. Y. R. Chen, "Integrated scheduling, allocation and module selection for design-space exploration in high-level synthesis," *IEE Proceedings - Computers and Digital Techniques*, vol. 142, no. 1, pp. 65–71, 1995.

[121] K. Ito, L. E. Lucke, and K. K. Parhi, "ILP-based cost-optimal DSP synthesis with module selection and data format conversion," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 6, no. 4, pp. 582–594, 1998.

[122] W. Sun, M. J. Wirthlin, and S. Neuendorffer, "FPGA Pipeline Synthesis Design Exploration Using Module Selection and Resource Sharing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 254–265, 2007.

[123] J. Cong, M. Huang, B. Liu, P. Zhang, and Y. Zou, "Combining module selection and replication for throughput-driven streaming programs," in *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2012, pp. 1018–1023.

[124] L. P. Carloni and A. L. Sangiovanni-Vincentelli, "Performance Analysis and Optimization of Latency Insensitive Systems," in *Proceedings of the 37th Annual Design Automation Conference*, ser. DAC '00. New York, NY, USA: Association for Computing Machinery, 2000, p. 361–367. [Online]. Available: https://doi.org/10.1145/337292.337441

[125] M. R. Casu and L. Macchiarulo, "A New Approach to Latency Insensitive Design," in *Proceedings of the 41st Annual Design Automation Conference*, ser. DAC '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 576–581. [Online]. Available: https://doi.org/10.1145/996566.996725

[126] M. Singh and M. Theobald, "Generalized latency-insensitive systems for single-clock and multi-clock architectures," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, vol. 2, 2004, pp. 1008–1013 Vol.2.

[127] R. L. Collins and L. P. Carloni, "Topology-Based Performance Analysis and Optimization of Latency-Insensitive Systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 12, pp. 2277–2290, 2008.

[128] Vitis HLS Coding Styles, 2022. [Online]. Available: https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/vitis_hls_coding_styles.html

[129] Optimization Techniques in Vitis HLS, 2022. [Online]. Available: https://www.xilinx.com/html_docs/xilinx2021_1/vitis_doc/vitis_hls_optimization_techniques.html

[130] J. Cheng, L. Josipović, P. Ienne, G. Constantinides, and J. Wickerson, "Combining Dynamic & Static Scheduling in High-level Synthesis," in *Proceedings of the 2020*

ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ser. FPGA '20. Monterey, CA: ACM, 2020.

[131] J. Cheng, J. Wickerson, and G. A. Constantinides, "Finding and Finessing Static Islands in Dynamically Scheduled Circuits," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 89–100. [Online]. Available: https://doi.org/10.1145/3490422.3502362

[132] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of Probabilistic Real-time Systems," in *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, ser. LNCS, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 585–591.

[133] M. Drazić, V. Kovacevic-Vujcić, M. Cangalović, and N. Mladenović, "Glob—a new VNS-based software for global optimization," in *Global optimization*. Springer, 2006, pp. 135–154.

[134] M. Silva, J. Júlvez, C. Mahulea, and C. R. Vázquez, "On fluidization of discrete event models: observation and control of continuous Petri nets," *Discrete Event Dynamic Systems: Theory and Applications*, vol. 21, no. 4, pp. 427–497, 2011.

[135] L. Ya. Rosenblum and A.V. Yakovlev, "Signal graphs: from self-timed to timed ones," in *Proc. of the Int. Workshop on Timed Petri Nets*. Torino, Italy: IEEE Computer Society Press, 1985, pp. 199–207.

[136] G. Chiola, M. A. Marsan, G. Balbo, and G. Conte, "Generalized stochastic Petri nets: a definition at the net level and its implications," *IEEE Transactions on Software Engineering*, vol. 19, no. 2, pp. 89–107, 1993.

[137] J. Júlvez and C. Mahulea, "SimHPN: a MATLAB toolbox for continuous Petri nets," *IFAC Proceedings Volumes*, vol. 43, no. 12, pp. 21–26, 2010, 10th IFAC Workshop on Discrete Event Systems. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1474667015324289

[138] R. David and H. Alla, *Discrete, Continuous and Hybrid Petri Nets.* Berlin: Springer, 2004, (2^{nd} edition, 2010).

[139] F. Balduzzi, A. Giua, and G. Menga, "First-order hybrid Petri nets: a model for optimization and control," *IEEE Transactions on Robotics and Automation*, vol. 16, no. 4, pp. 382–399, 2000.

[140] R. M. Shapiro, "Validation of a VLSI chip using hierarchical colored Petri nets," *Microelectronics Reliability*, vol. 31, no. 4, pp. 607 – 625, 1991. [Online]. Available: http://www.sciencedirect.com/science/article/pii/002627149190006S

[141] K. L. McMillan, "Using Unfoldings to Avoid the State Explosion Problem in the Verification of Asynchronous Circuits," in *Proceedings of the Fourth International Workshop on Computer Aided Verification*, ser. CAV '92. Berlin, Heidelberg: Springer-Verlag, 1992, p. 164–177.

[142] J. Carmona, J. Cortadella, and E. Pastor, "A Structural Encoding Technique for the Synthesis of Asynchronous Circuits," vol. 50, 01 2001, pp. 157–166.

[143] J.-I. Rocha, O. Páscoa Dias, and L. Gomes, "Improving synchronous dataflow analysis supported by petri net mappings," *Electronics*, vol. 7, no. 12, 2018. [Online]. Available: https://www.mdpi.com/2079-9292/7/12/448

[144] C. V. Ramamoorthy and G. S. Ho, "Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 5, pp. 440–449, 1980.

[145] A. Xie and P. A. Beerel, *Performance Analysis of Asynchronous Circuits and Systems Using Stochastic Timed Petri Nets.* Boston, MA: Springer US, 2000, pp. 239–268. [Online]. Available: https://doi.org/10.1007/978-1-4757-3143-9_13

[146] B. R. T. M. Witlox, P. van der Wolf, E. H. L. Aarts, and W. M. P. van der Aalst, *Performance Analysis of Dataflow Architectures Using Timed Coloured Petri Nets.* Boston, MA: Springer US, 2000, pp. 269–289. [Online]. Available: https://doi.org/10.1007/978-1-4757-3143-9_14

[147] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[148] L. Recalde, E. Teruel, and M. Silva, "Autonomous Continuous P/T Systems," in *Application and Theory of Petri Nets 1999*. Springer Berlin Heidelberg, 1999, pp. 107–126.

[149] J. Cheng, J. Wickerson, and G. A. Constantinides, "Probabilistic Scheduling in High-Level Synthesis," in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2021, pp. 195–203.

[150] C. E. Leiserson, F. M. Rose, and J. B. Saxe, "Optimizing synchronous circuitry by retiming (preliminary version)," in *Third Caltech conference on very large scale integration*. Springer, 1983, pp. 87–116.

[151] V. Sarkar, "Optimized unrolling of nested loops," in *Proceedings of the 14th international conference on Supercomputing*, 2000, pp. 153–166.

[152] Y. Markovskiy and Y. Patel, "Simple Symmetric Multithreading in Xilinx FPGAs," 2002.

[153] N. Weaver, Y. Markovskiy, Y. Patel, and J. Wawrzynek, "Post-Placement C-Slow Retiming for the Xilinx Virtex FPGA," in *Proceedings of the 2003 ACM/SIGDA Eleventh International Symposium on Field Programmable Gate Arrays*, ser. FPGA '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 185–194. [Online]. Available: https://doi.org/10.1145/611817.611845

[154] J. Cheng, J. Wickerson, and G. A. Constantinides, "Dynamic C-Slow Pipelining for HLS," in *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2022, pp. 1–10.

[155] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 101–113. [Online]. Available: https://doi.org/10.1145/1375581.1375595

[156] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet, "Polly-Polyhedral optimization in LLVM," in *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, vol. 2011, 2011, p. 1.

[157] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[158] D. R. Butenhof, *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

[159] J. Liu, J. Wickerson, and G. A. Constantinides, "Loop Splitting for Efficient Pipelining in High-Level Synthesis," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2016, pp. 72–79.

[160] D. Cabrera, X. Martorell, G. Gaydadjiev, E. Ayguade, and D. Jiménez-González, "OpenMP extensions for FPGA accelerators," in *2009 International Symposium on Systems, Architectures, Modeling, and Simulation*. IEEE, 2009, pp. 17–24.

[161] Y. Leow, C. Ng, and W. Wong, "Generating hardware from OpenMP programs," in *2006 IEEE International Conference on Field Programmable Technology*, 2006, pp. 73–80.

[162] J. Choi, S. Brown, and J. Anderson, "From software threads to parallel hardware in high-level synthesis for FPGAs," in *2013 International Conference on Field-Programmable Technology (FPT)*, 2013, pp. 270–277.

[163] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK: a high-level synthesis framework for applying parallelizing compiler transformations," in *16th International Conference on VLSI Design, 2003. Proceedings.*, 2003, pp. 461–466.

[164] Intel FPGA SDK for OpenCL Software Technology, 2021. [Online]. Available: https://www.intel.co.uk/content/www/uk/en/software/programmable/sdk-for-opencl/overview.html

[165] K. Kennedy and K. S. McKinley, "Maximizing loop parallelism and improving data locality via loop fusion and distribution," in *Languages and Compilers for Parallel Computing*,

U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 301–320.

[166] L. Josipović, A. Marmet, A. Guerrieri, and P. Ienne, "Resource Sharing in Dataflow Circuits," in *Proceedings of the 30th IEEE Symposium on Field-Programmable Custom Computing Machines*, New York, May 2022, to appear.

[167] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: A compiler infrastructure for the end of Moore's law," *arXiv preprint arXiv:2002.11054*, 2020.

[168] W. S. Moses, L. Chelini, R. Zhao, and O. Zinenko, "Polygeist: Raising C to Polyhedral MLIR," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '21. New York, NY, USA: Association for Computing Machinery, 2021.

[169] Torch-MLIR, 2023. [Online]. Available: https://github.com/llvm/torch-mlir

[170] L. Guo, Y. Chi, J. Wang, J. Lau, W. Qiao, E. Ustun, Z. Zhang, and J. Cong, "AutoBridge: Coupling Coarse-Grained Floorplanning and Pipelining for High-Frequency HLS Design on Multi-Die FPGAs," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 81–92. [Online]. Available: https://doi.org/10.1145/3431920.3439289

[171] L. Guo, P. Maidee, Y. Zhou, C. Lavin, J. Wang, Y. Chi, W. Qiao, A. Kaviani, Z. Zhang, and J. Cong, "RapidStream: Parallel Physical Implementation of FPGA HLS Designs," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1–12. [Online]. Available: https://doi.org/10.1145/3490422.3502361

[172] C. Paulin-Mohring, "Inductive definitions in the system Coq rules and properties," in *International Conference on Typed Lambda Calculi and Applications*. Springer, 1993, pp. 328–345.

[173] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[174] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, p. 99–110, aug 2013. [Online]. Available: https://doi.org/10.1145/2534169.2486011

[175] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang, "Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 766–780.

[176] L. Song, Y. Chi, A. Sohrabizadeh, Y.-k. Choi, J. Lau, and J. Cong, "Sextans: A streaming accelerator for general-purpose sparse-matrix dense-matrix multiplication," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 65–77. [Online]. Available: https://doi.org/10.1145/3490422.3502357