EPFL

Thèse n° 7211

High-Level Synthesis of Dynamically Scheduled Circuits

Présentée le 27 août 2021

Faculté informatique et communications
Laboratoire d'architecture des processeurs
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Lana JOSIPOVIĆ

Acceptée sur proposition du jury

Prof. B. Falsafi, président du jury
Prof. P. Ienne, directeur de thèse
Dr D. Burger, rapporteur
Prof. J. Cong, rapporteur
Prof. J. Cortadella, rapporteur
Prof. G. De Micheli, rapporteur

École polytechnique fédérale de Lausanne

2021

Acknowledgments

I would like to express my gratitude to all those who made this thesis possible and who had a significant impact on this great chapter of my life.

First and foremost, I am deeply grateful to my PhD advisor, Paolo Ienne, who introduced me to the world of research and profoundly impacted both my personal and my professional development. His constant feedback, encouragement, and enthusiasm continue to motivate and inspire me to achieve my best. I truly could not have wished for a better mentor to guide me through my PhD journey.

I would like to thank Doug Burger, Jason Cong, Jordi Cortadella, and Giovanni De Micheli for serving on my thesis jury, as well as Babak Falsafi for presiding it. Their insightful comments and questions have significantly improved this thesis. I am also very grateful for the tremendous support and assistance they provided me in defining my future career steps.

I am thankful that I had the opportunity to collaborate and learn from many inspiring and talented people. A very special thanks again to Jordi Cortadella, whose expertise has greatly influenced this thesis and whose continuous support throughout my PhD has enabled me to advance and grow. I had the pleasure of collaborating with George Constantinides, whom I especially thank for his valuable advice and his kind support of my career development. I would also like to extend my gratitude to Steve Neuendorffer for helping me to expand my skills and knowledge, as well as for his friendliness and hospitality during my stay in California.

I thank all the members of the FPGA community that I had the pleasure of meeting and discussing with throughout the years, as well as all my colleagues and friends at Xilinx and Microsoft Research who made my internship experiences unforgettable. I am thankful to all the students and collaborators who contributed to my research and the work described in this thesis.

I was very fortunate to work alongside many current and former LAP members, including Ana, André, Andrea, Andrew, Aya, Chantal, David, Dewmini, Grace, Jovan, Mikhail, Nithin, René, Sahand, and Stefan. I would especially like to thank Andrea for his significant work on Dynamatic, Chantal for helping me survive in French-speaking Switzerland, David for motivating me to pursue a PhD, and Sahand for his constant availability and willingness to help out. Finally, my biggest thanks goes to Grace, whose amazing support, extreme caring, and many hugs made a huge difference in my PhD life.

Abstract

I am grateful to everyone from EDIC who started the PhD journey with me and stayed close throughout, as well as all the EPFL friends I made along the way, for making my PhD incredibly fun and enjoyable. Special mentions go to Dušan, Helena, and Hermina for making Lausanne feel a bit closer to home, as well as George, Lefteris, and Marios for sharing my skiing enthusiasm and always being up for a day on the slopes. An extra thanks to Lefteris for being a great neighbor and the person that I could always count on. Many thanks to Endri for the weekly coffees and long conversations, as well as to Javier for making TAing surprisingly fun.

Next, I would like to thank all my Croatian friends who enthusiastically cheered me on along the way, including Ami, Dragec, Karla, Kuna, Matea, Petra, Tea, and Tena. I very much appreciate that we managed to stay in touch and that we find time to get together whenever I come home. A very special thanks to Tea and Tena for their lifelong friendship and for being there for me no matter what. Above all, I thank Marko for his endless patience, strong devotion, and continuous encouragement during the past years; I am extremely grateful for having him by my side.

Last, but in no way least, I would like to thank my family: my parents Tatjana and Ivo, and my grandparents Mimi, Zora, and Ante, for enabling me to achieve all my goals with their unconditional love and support.

Lausanne, January 18th, 2021 Lana Josipović

Abstract

High-Level Synthesis (HLS) tools generate hardware designs from high-level programming languages. These tools almost universally build datapaths that are controlled using a *centralized controller* which relies on a *static, compile-time schedule* to determine the cycle when each operation executes. Such an approach results in high-throughput pipelines in cases where memory accesses *are provably independent* and critical control decisions *are determinable* during code compilation. Unfortunately, when this is not the case, the tools must make pessimistic assumptions, yielding inferior schedules and lower performance. An alternative HLS approach is to create *dataflow circuits* out of high-level code. Dataflow circuits are built out of units which communicate using point-to-point pairs of handshake control signals; data is propagated from unit to unit as soon as memory and control dependences allow it and stalled by the handshaking mechanism otherwise. This distributed control mechanism effectively implements a *dynamic schedule*, where scheduling decisions are made locally in the circuit as it runs, hence achieving behaviors which are beyond the capabilities of statically scheduled circuits.

Although translating high-level code into dataflow circuits seems relatively straightforward, a naive translation is not sufficient to achieve functional correctness, high performance, and area efficiency. Firstly, without appropriate buffer placement and sizing, dataflow circuits exhibit only limited pipelining capabilities. Secondly, in the absence of a static schedule, resource sharing opportunities are difficult to identify; in addition, sharing may cause deadlock and compromise the functionality of the circuit. Thirdly, memory accesses in a dataflow circuit may execute in an order different than the one specified in the original program—a naive memory interface is not always sufficient to guarantee that all memory dependences are honored. Finally, standard dataflow circuits do not support speculation, i.e., the ability to execute some operations before it is certain whether they are correct or required, which prevents pipelining when a memory or a control dependence takes a long time to resolve.

The contribution of this thesis is to develop techniques that make dataflow circuits truly competitive in the HLS context. We first present a complete set of rules and transformations to create dataflow circuits out of high-level specifications (i.e., C/C++ programs). We detail a methodology to systematically place and size buffers in dataflow circuits to achieve high-throughput pipelines. We show how to automatically identify performance-acceptable resource sharing opportunities and describe a sharing mechanism which achieves functionally correct and deadlock-free dataflow designs. We detail the construction of a memory interface (i.e., a load-store queue)

for dataflow circuits that can correctly handle memory accesses arriving out of order and show how to automatically customize this interface to a particular application. Further, we present a generic framework for handling speculation in dataflow circuits. Finally, we show that these techniques can reap significant area/performance benefits in appropriate situations.

All these features enable dataflow circuits to achieve dynamic behaviors similar to those of modern superscalar processors; we believe that these behaviors are key for HLS to be successful in new contexts and broader application domains.

Keywords: high-level synthesis, dataflow circuits, dynamic scheduling.

Résumé

Des outils de *Synthèse de haut-niveau* (HLS) peuvent générer des circuits électroniques à partir d'un language de programmation haut-niveau. Ces outils créent quasiment toujours un chemin de données contrôlé à partir d'un *controlleur centralisé* utilisant un *ordonnancement statique connu au temps de compilation* pour déterminer le cycle pendant lequel chaque opération s'exécute. Une telle approche permet de créer des pipelines ayant une bande passante élevée dans les cas où les accès à la mémoire *sont prouvablement indépendants* et que des décisions de contrôle critiques *sont déterministiques* pendant la compilation du code. Malheureusement, quand ce n'est pas le cas, les outils doivent faire des hypothèses pessimistes, créant ainsi des ordonnancements de plus basse qualité et une performance réduite. Une solution HLS alternative est de créer des *circuits dataflow* à partir de code de haut niveau. Les circuits dataflow sont conçus à partir d'unités qui communiquent en utilisant des signaux d'établissement de liaison (handshake) ; les données se propagent d'unité à unité aussitôt que les dépendances de contrôle et de mémoire le permettent, et sont arrêtés par le mechanisme de handshake le cas échéant. Ce mechanisme de contrôle distribué implémente ainsi un *ordonnancement dynamique* où les décisions d'ordonnancement sont faites de façon local dans le circuit pendant qu'il s'exécute. De tels circuits peuvent ainsi atteindre des comportements qui dépassent les capacités de circuits ordonnancés de façon statique.

Bien que la traduction de code haut-niveau en circuits dataflow est relativement simple, une traduction naïve n'est pas suffisante pour atteindre un fonctionnement correct, une haute performance, ainsi qu'un bon rendement au niveau du surface nécessaire pour concevoir le circuit. Premièrement, sans un bon placement et dimensionnement de mémoires tampon, les circuits dataflow n'auraient qu'une capacité limitée pour être pipeliné. Deuxièement, en l'absence d'un ordonnancement statique, il est difficile de détecter des opportunités pour partager des ressources. De plus, un tel partage de ressources pourrait causer un interblocage et ainsi compromettre la fonctionalité du circuit. Troisièmement, les accès à la mémoire dans un circuit dataflow peuvent s'exécuter dans un ordre différent que celui spécifié dans le programme original — une interface mémoire naïve n'est parfois pas suffisante pour guarantir que toutes les dépendances des accès à la mémoire sont honorées. Finalement, les circuits dataflow standards ne supportent pas de spéculation, c'est à dire la capacité d'exécuter certaines opérations avant qu'il ne soit certain qu'elles soient necessaires, ce qui empêche le pipelining quand une dépendance de donnée ou de contrôle prend du temps à résoudre.

Abstract

La contribution de cette thèse est de développer des techniques qui permettent aux circuits dataflow d'être véritablement compétitifs dans le contexte de la HLS. Nous présentons en premier un ensemble de règles et de transformations pour créer des circuits dataflow à partir de spécifications de haut-niveau telles que des programmes C/C++. Nous détaillons ensuite une méthodologie pour placer et dimensionner des mémoires tampons dans des circuits dataflow afin de créer des pipelines à bande passante élevée. Nous montrons comment identifier de façon automatique des opportunités de partage de ressources ayant des performances acceptables et décrivons un mécanisme de partage pour créer des circuits dataflow étant fonctionnellement corrects et sans interblocage. Nous détaillons la construction d'une interface mémoire (une *load-store* queue) pour des circuits dataflow qui peut traîter des accès à la mémoire arrivant dans le désordre et montrons comment adapter cette interface à une application spécifique. De plus, nous présentons un cadre générique pour traîter la spéculation dans les circuits dataflow. Finalement, nous montrons que ces techniques permettent de gagner significativement plus de performance et de rendement de surface dans des situations appropriées.

Toutes ces fonctionnalités permettent aux circuits dataflow d'exposer un comportement dynamique similaire à celui des processeurs superscalaires modernes. Nous pensons que ces nouvelles techniques sont la clé pour le succès de la HLS dans de nouveaux contextes ainsi que dans des domaines d'application plus large.

Mots clefs : synthèse de haut-niveau, circuits dataflow, ordonnancement dynamique.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

The slowdown in transistor scaling and the end of Moore's law signal a clear need to invest in new computing paradigms [105]. Specialized hardware devices, such as *Field Programmable Gate Arrays* (FPGAs) and *Application-Specific Integrated Circuits* (ASICs), are a promising solution to achieve high processing capabilities and energy efficiency; recently, these devices have been customized for large-scale AI applications [78, 49], integrated into datacenters to accelerate massive data [23, 3], and packaged with processors for high parallelism [29]. However, a major barrier to the global success of specialized computing is the difficulty of hardware design. *High-Level Synthesis* (HLS) tools can generate hardware designs from high-level programming languages and should liberate designers from the details of hardware description languages like VHDL and Verilog. Despite their progress and some commercial success in the last decade, HLS tools still tend to be criticized for the difficulty of extracting the desired level of performance: generating good circuits from high-level languages still requires peculiar code restructuring, expert user interaction, and extensive experimentation with the tools [72]. Moreover, current HLS techniques face a fundamental issue when handling irregular applications: because they rely on static scheduling, i.e., the cycle in which each operation executes is fixed at compile time [47], they force worst-case assumptions on memory and control dependences. Therefore, HLS tools are still usable only by designers with hardware expertise and acceptable only for some classes of applications. If specialized computing is to play a key role in dealing with the increasing computational demands in the post-Moore era, it is imperative that hardware design becomes accessible to a variety of users from different application domains.

1.1 The Limitations of Today's HLS

Circuits produced by HLS tools are typically built out of datapaths that are controlled using a preplanned, central controller. The controller relies on a static schedule, fixed at compile time, to determine the clock cycle when each operation can execute. Such an approach is effective in regular code where compile-time information is sufficient to obtain a high-throughput, pipelined schedule. However, when the code contains unpredictable memory or control de-

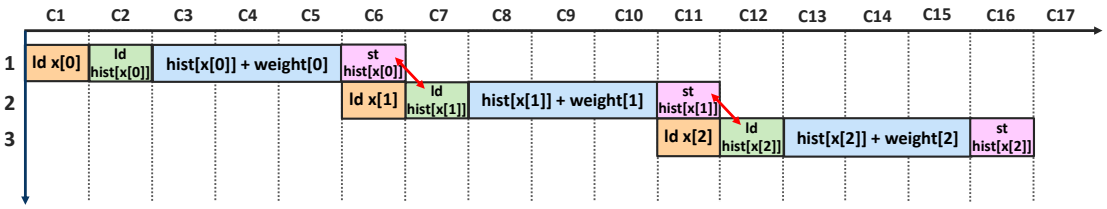
pendences, long-latency control decisions, or variable-latency events, the tool must make pessimistic assumptions, yielding inferior schedules and lower performance.

Unpredictable memory dependences. The example in Figure 1.1a illustrates the behavior of standard HLS approaches when memory access patterns cannot be determined at compile time. In this loop, there is a *possible* read-after-write dependence between the memory read of `hist[x[i+1]]` and the memory write to `hist[x[i]]` of the previous iteration. There is intrinsically no way a compiler or an HLS tool can determine whether such a dependence exists, nor is it, in general, possible for a programmer to help the tool: in practice, the read may seldom or even never address the same value just written in memory, but there is no way to exclude a priori that this might happen. Ultimately, any HLS technique based on static scheduling hits the problem of potential dependences and needs to account for the worst-case scenario, irrespective of the actual data fetched from memory. The result is a conservative schedule, valid for any possible input values, which assumes a dependence in every loop iteration and postpones the read until the previous, possibly dependent write has been completed [70].

Unpredictable control flow. A similar issue occurs in the presence of unpredictable control flow. The loop in Figure 1.1b has a control flow decision (`if`) which depends on the actual data being read from arrays `a[]` and `b[]`. The operation which might take place in a specific iteration (`s += d`) introduces a dependence between iterations and delays the next iteration whenever the condition is true. When pipelining this loop, a typical HLS tool needs to create a conservative execution plan for the various operations in the loop which is valid in every possible case. The resulting schedule is shown in the figure: although the condition is true only for the second and third iteration, a "space" is reserved in the schedule as if the condition were true everywhere, hence limiting the pipelining of the loop.

Long-latency control flow decisions. Although static scheduling supports predication and if-conversion, these techniques are not applicable to every performance-critical control decision and are limited in the presence of memory accesses or complex control flow. A simple example is illustrated in Figure 1.1c: the condition on the loop continuation takes multiple cycles to compute and a new loop iteration can start only after this condition has been determined, hence completely preventing loop pipelining. This conservatism is due to the limited ability of a static schedule to adapt to different outcomes and to revert the state in case of a failed prediction.

Variable-latency events. Standard HLS techniques suffer in the presence of variable-latency operations or memory accesses: a typical example is that of data accesses from an external shared memory, where latencies may vary due to memory system hierarchy or system-level contentions. In such situations, static HLS techniques typically need to stall the complete pipeline when a long-latency event (e.g., a cache miss) occurs; alternatively, they resort to adding excessive pipeline stages to accommodate for the worst-case delay [108]. These performance- and area-degrading solutions are a clear artifact of the inability of static scheduling to accommodate dynamic events.

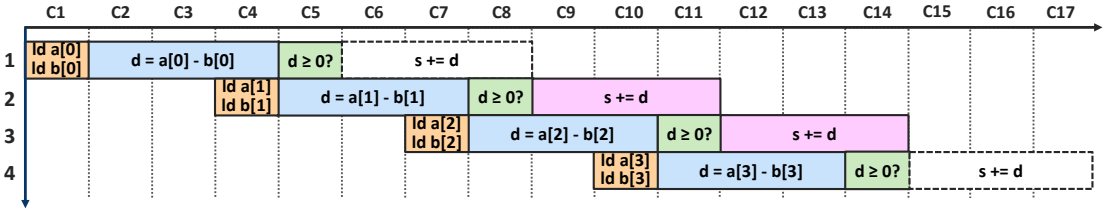



```
for (i=0; i<N; i++) {
   hist[x[i]] = hist[x[i]] + weight[i];
}
```

```
1: x[0]=5 → ld hist[5]; st hist[5];
2: x[1]=4 → ld hist[4]; st hist[4];
3: x[2]=4 → ld hist[4]; st hist[4];
```

RAW

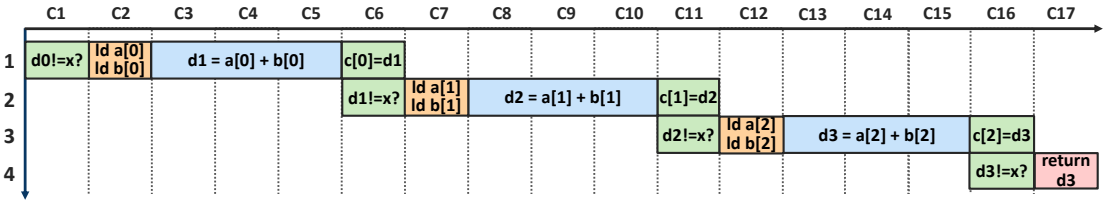
(a) Code with memory dependences undeterminable at compile time and its static schedule.




```
float d, s = 0.0; int i;
for (i=0; i<100; i++){
   d = a[i] - b[i];
   if (d >= 0)
      s += d;
}
```

```
a[0]=1.0; b[0]=3.0;
a[1]=4.0; b[1]=3.0;
a[2]=2.0; b[2]=2.0;
a[3]=4.0; b[3]=5.0;
        ⋮
```

(b) Code with control flow undeterminable at compile time and its static schedule.




```
float d=0.0; x=100.0; int i=0;
while (d<x) do {
   d = a[i] + b[i];
   c[i] = d;
   i++; }
return d;
```

```
1: a[0]=50.0; b[0]=30.0
2: a[1]=40.0; b[1]=40.0
3: a[2]=50.0; b[2]=60.0 → exit
```

(c) Code with a long-latency control decision and its static schedule.

Figure 1.1 – Limitations of standard HLS. In the presence of unpredictable memory accesses, control flow, or long-latency control flow decisions, static HLS tools create a conservative schedule, resulting in lower performance.

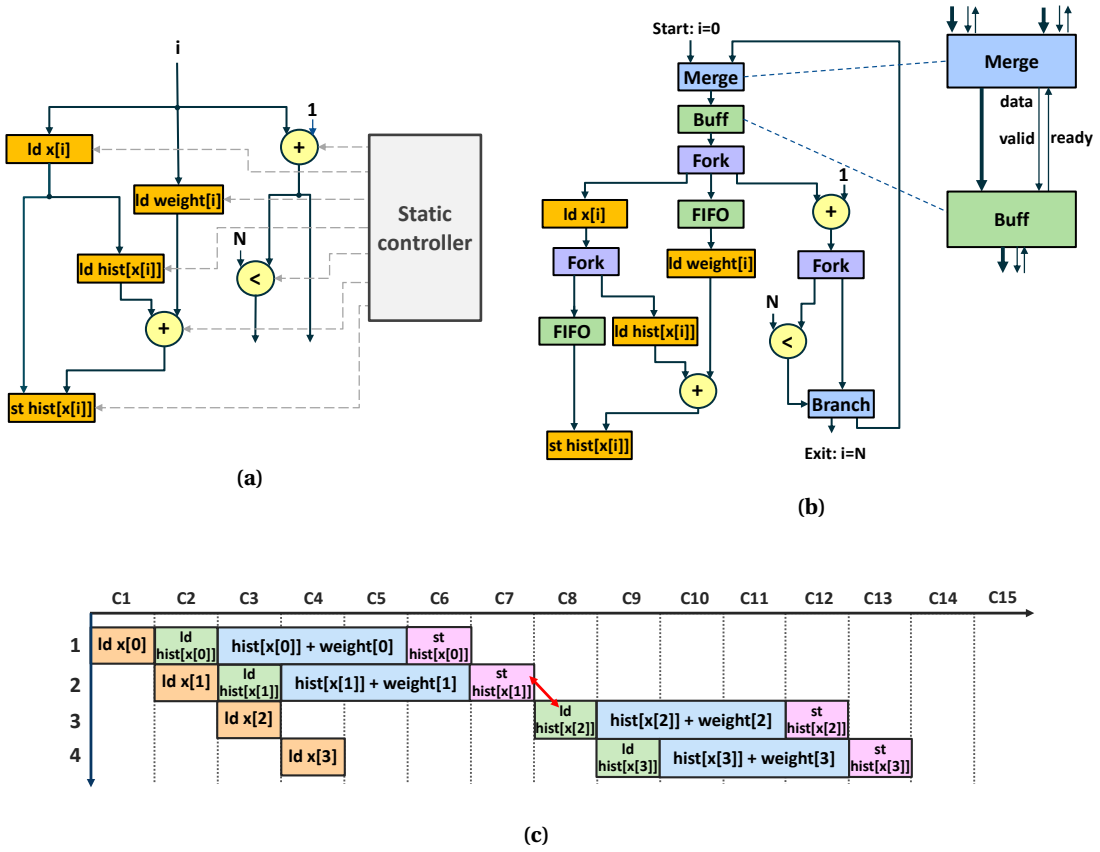


Figure 1.2 – A statically (Figure 1.2a) and a dynamically (Figure 1.2b) scheduled circuit. The static circuit has a preplanned controller which determines the time when each operation will execute; it implements the conservative schedule from Figure 1.1a. In contrast, the dynamically scheduled circuit contains a distributed control system which enables decision-making at runtime and offers greater flexibility and performance, achieving the schedule in Figure 1.2c.

1.2 A Completely Different Way to Do HLS

The key to avoid the limitations of static scheduling is to refrain from triggering the operations through a centralized preplanned controller (Figure 1.2a), but to make scheduling decisions locally in the circuit as it runs: as soon as all conditions for execution are satisfied (e.g., the operands are available or critical control decisions are resolved), an operation starts. *Dataflow circuits* [37] are a natural method to realize such behavior. Such circuits are built out of units that implement *latency-insensitivity* by communicating with their predecessors and successors through point-to-point pairs of handshake control signals, which indicate the availability of a new piece of data from the source unit and the readiness of the target unit to accept it (Figure 1.2b). The data is propagated from unit to unit as soon as memory and control dependences allow it—otherwise, the handshaking mechanism stalls the data on-the-fly. This distributed control mechanism effectively implements a *dynamic schedule*, such as the one in Figure 1.2c:

when a read-after-write dependence exists (in this case, between the second and the third iteration) the dynamically scheduled circuit will stall the pipeline to prevent a hazard. Otherwise, in the absence of an address collision, it will start a new iteration on every cycle and gain, in this case, up to a factor 5 in performance. Similar dynamic behaviors are achievable in the other situations from Figure 1.1—we will discuss them in the following chapters. These behaviors are beyond what classic static techniques can achieve.

1.3 Computer Architects Have Been There Already

The contrast between static and dynamic scheduling in HLS is in line with the experience in computer architecture with *Very Long Instruction Processors* (VLIWs) and *superscalar out-of-order* processors [64].

In VLIWs, the problem of deciding when instructions can be executed is left completely to the compiler: the hardware simply fetches at once and executes groups of operations which can be performed together. The program is effectively a schedule computed statically by the compiler, exactly as in the case of statically scheduled HLS. In contrast, superscalar processors rely on complex mechanisms to achieve out-of-order behavior: *reservation stations* hold back fetched and decoded instructions until all of their operands are available; while some instructions are delayed due to a memory dependence or a cache miss, others can execute. Thanks to the *reorder buffer*, these processors can execute instructions speculatively when the outcome of a preceding branch is unknown or the existence of a memory dependence has not yet been ascertained. As with dynamically scheduled HLS, there is no schedule planned in advance: the schedule develops dynamically as operands become available.

While complex compilation techniques have been developed for VLIWs to exploit instruction-level parallelism (often requiring either complex heuristics to drive the optimization or pragmas from the programmers), dynamically scheduled out-of-order processors are capable of achieving good levels of parallelism on-the-fly and without extensive code preparation. In fact, many of the key transformations to exploit fine-grain parallelism between operators in statically scheduled HLS derive from VLIW compilation techniques [82, 103]. Exactly as VLIWs, statically scheduled HLS suffers when handling code with irregular memory or control dependences; they are primarily successful in markets with extremely regular and predictable applications and where it is acceptable to tune code manually.

The dichotomy in computer architecture may tell us something about the future of dynamically scheduled HLS. With FPGAs moving to datacenters and facing broader application classes, HLS tools might have to satisfy the needs of general-purpose markets as well. Apart from the advantage of exploiting parallelism in cases where static scheduling cannot, the ability of dynamic scheduling to find an acceptable solution without the programmer's help may be critical in a future where HLS will not be driven by hardware designers (available to study the generated circuits and to restructure the input code) but by higher-level code generation tools (e.g., Delite [52]) and, ultimately, by software programmers.

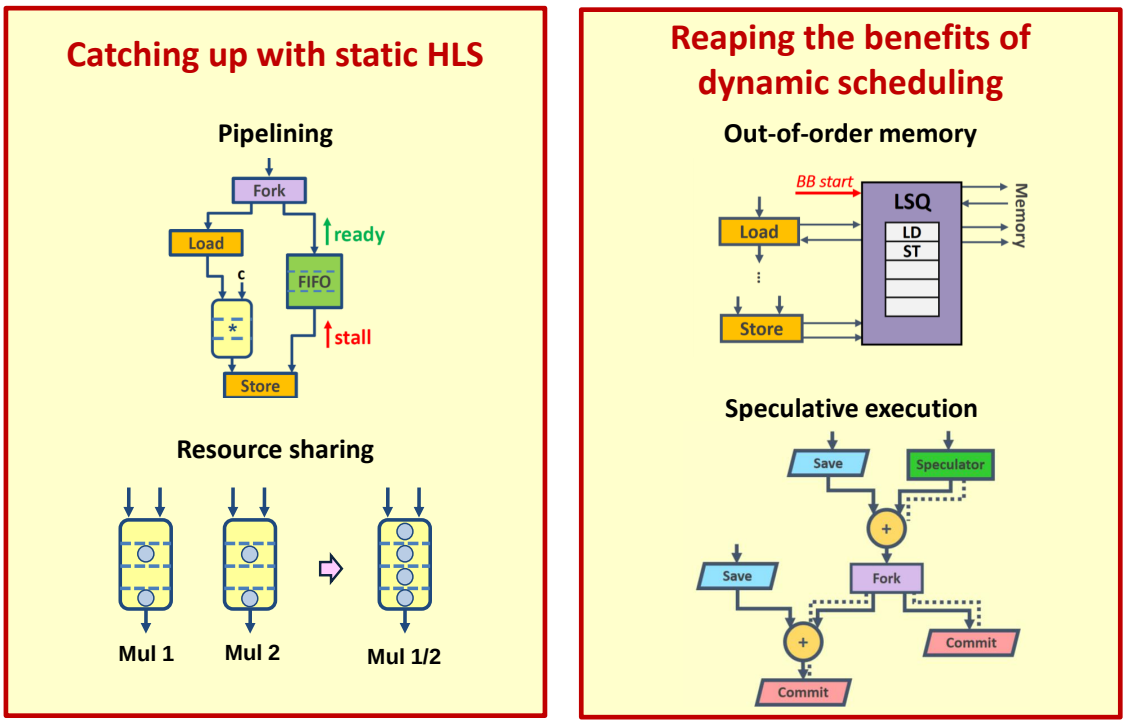


Figure 1.3 – Thesis contribution. We first describe optimizations which make dataflow circuits competitive with static HLS circuits (i.e., pipelining and resource sharing); then, we implement behaviors which are beyond the capabilities of classic HLS tools (i.e., out-of-order memory accesses and speculation).

1.4 Thesis Contribution

In this thesis, we pursue a form of HLS which produces dynamically scheduled, dataflow circuits out of high-level code. However, a straightforward translation is not sufficient to obtain circuits which are truly competitive and useful in the HLS context. The main contribution of this thesis, outlined in Figure 1.3, is to enhance dataflow circuits with the capabilities which are beyond those achievable by naively obtained dataflow circuits and existing HLS techniques. We first describe methodologies to exploit the same optimization opportunities that standard HLS relies on (i.e., pipelining and resource sharing), hence achieving circuits competitive to those created with standard HLS techniques. We then use the properties of dataflow circuits to achieve characteristics that standard HLS cannot support (i.e., out-of-order memory access execution and speculation). The resulting behaviors are similar to those of modern superscalar processors and achieve solutions which are, in particular cases, superior to statically scheduled HLS circuits: similarly to the tradeoff between VLIW processors and superscalars, the performance of demanding applications is very significantly improved at an affordable cost.

2 Dynamically Scheduled High-Level Synthesis

In this chapter, we discuss standard scheduling techniques and we explore in detail an example of a situation where dynamic extraction of operation-level parallelism proves essential for performance. We then present our methodology to automatically generate dynamically scheduled circuits from C code. Our approach borrows several ideas from the asynchronous domain, but produces perfectly synchronous designs which are directly comparable to standard HLS techniques. We conclude this chapter with a summary of challenges in achieving high-performance, area-efficient dataflow circuits and highlight the features that we will explore in the following chapters.

2.1 How Does Classic HLS Work?

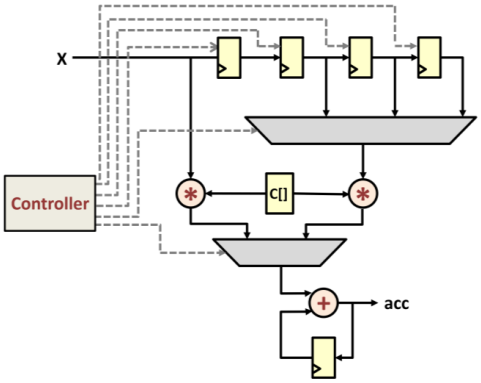
Hardware description languages (HDLs), such as VHDL and Verilog, have been used in the electronic design industry for decades to specify the details of hardware design in terms of low-level building blocks such as gates, registers, and multiplexers [8]. However, this description level requires hardware expertise and, typically, a longer time to develop the design. High-level synthesis tools allow designers to work at a higher level of abstraction by using a software language to specify the hardware functionality. This approach enables software engineers to program hardware and helps hardware engineers to speed up the design process as well as to efficiently explore the design space [92]. Although HLS can benefit both ASIC and FPGA designers, HLS tools are particularly gaining popularity in the FPGA domain, as the programming challenges are one of the biggest barriers to the mainstream adoption of these devices [8, 33].

Different HLS tools rely on various high-level representations to describe the underlying hardware; the most popular ones use C/C++ as an input language [119, 18]. Generally speaking, the user provides the input functional specification and particular design constraints such as the target device, desired clock frequency, and memory interface description; the tool then automatically analyzes concurrency, inserts registers to achieve the specified frequency, generates

This chapter is based on the work published at the *26th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2018 [73].

a. No optimization

```
acc = 0;
for (i = 3; i >= 0; i--) {
   if (i == 0) {
      shift_reg[0] = x;
      acc += x * c[0];
   } else {
      shift_reg[i] = shift_reg[i-1];
      acc += shift_reg[i] * c[i];
   }
}
y = acc;
```

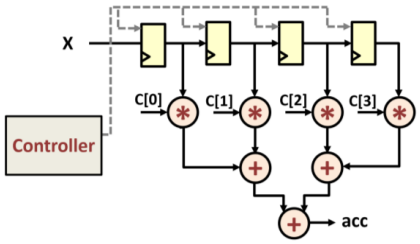


b. Unrolling

```
shift_reg[3] = shift_reg[2];
shift_reg[2] = shift_reg[1];
shift_reg[1] = shift_reg[0];
shift_reg[0] = x;

acc = shift_reg[3] * c[3];
acc += shift_reg[2] * c[2];
acc += shift_reg[1] * c[1];
acc += shift_reg[0] * c[0];

y = acc;
```



c. Pipelining

```
shift_reg[3] = shift_reg[2];
shift_reg[2] = shift_reg[1];
shift_reg[1] = shift_reg[0];
shift_reg[0] = x;

acc = 0;
for (i = 10; i >= 0; i--) {
   #pragma HLS pipeline
   acc += shift_reg[i] * c[i];
}
y = acc;
```

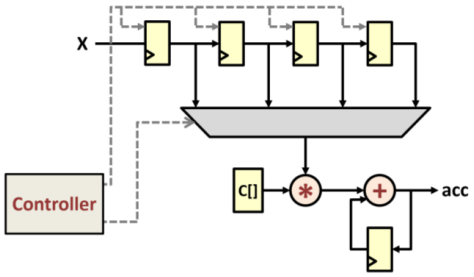


Figure 2.1 – Design space exploration with static HLS [80]. All three codes in the figure describe the same functionality (i.e., an FIR filter); yet, the resulting HLS solutions differ in area and performance.

the control and datapath logic, and maps data onto storage elements to optimize the bandwidth and resource usage [80]. The user is typically required to restructure the code and annotate it with pragmas to guide the tool in reaching the desired design point.

Figure 2.1 illustrates several possibilities to specify the functionality of a simple FIR filter in C code as well as the resulting circuits produced by an HLS tool [80]; apart from the different datapaths, as shown in the figure, each design has a kernel-specific controller which triggers the datapath components at appropriate clock cycles; we will discuss this functionality in the following section. The first circuit is obtained from a typical software representation, without any hardware-specific annotations or code restructuring. The second code is manually unrolled to explicitly express available parallelism to the HLS tool—as the corresponding circuit suggests, this design will employ multiple operators which can be used concurrently. The third design point uses a pragma to indicate to the tool that the code should be pipelined, i.e., the loop

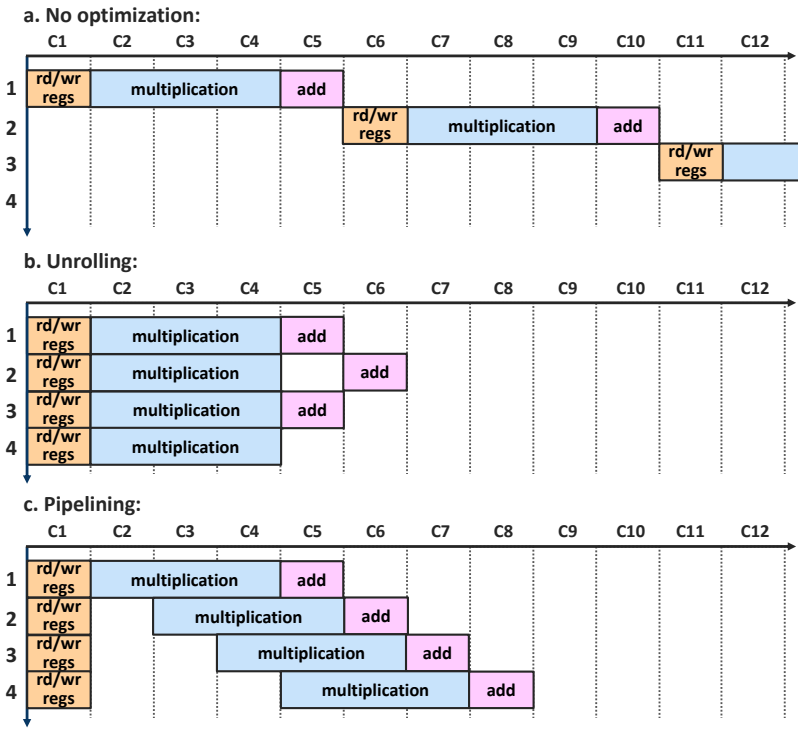


Figure 2.2 – The schedules of the three design points from Figure 2.1.

iterations should overlap for performance benefits. It is evident from the figure that the circuits differ in the number of employed resources (i.e., adders, multipliers, multiplexers); they also differ in performance, as we will discuss next.

2.1.1 Scheduling in HLS

HLS relies on a series of compiler optimizations to achieve performance- and area-efficient designs; some techniques are exploited by compilers in general (e.g., code motion, if-conversion), whereas others are hardware-specific (e.g., bitwidth analysis, operation chaining). One of the key algorithms in HLS synthesis is *scheduling*, i.e., deciding the clock cycle in which each operation will execute. It is typically achieved through *systems of difference constraints* (SDC) modeling which incorporates a variety of constraints, such as resource usage, data dependences, control dependences, and clock frequency [17, 34, 120].

The three schedules in Figure 2.2 correspond to the circuits in Figure 2.1. The schedule of each design is regulated by the controller; it implements a *finite state machine* (FSM) which controls the behavior of the datapath by triggering operations, enabling registers, and multiplexing values in appropriate clock cycles. The first schedule corresponds to the sequential execution of the software code: one iteration starts after the previous one has completed. The code restructuring in the second figure enables operations to execute in parallel, hence lowering the execution time, but with the investment of additional resources. The third circuit employs loop pipelining:

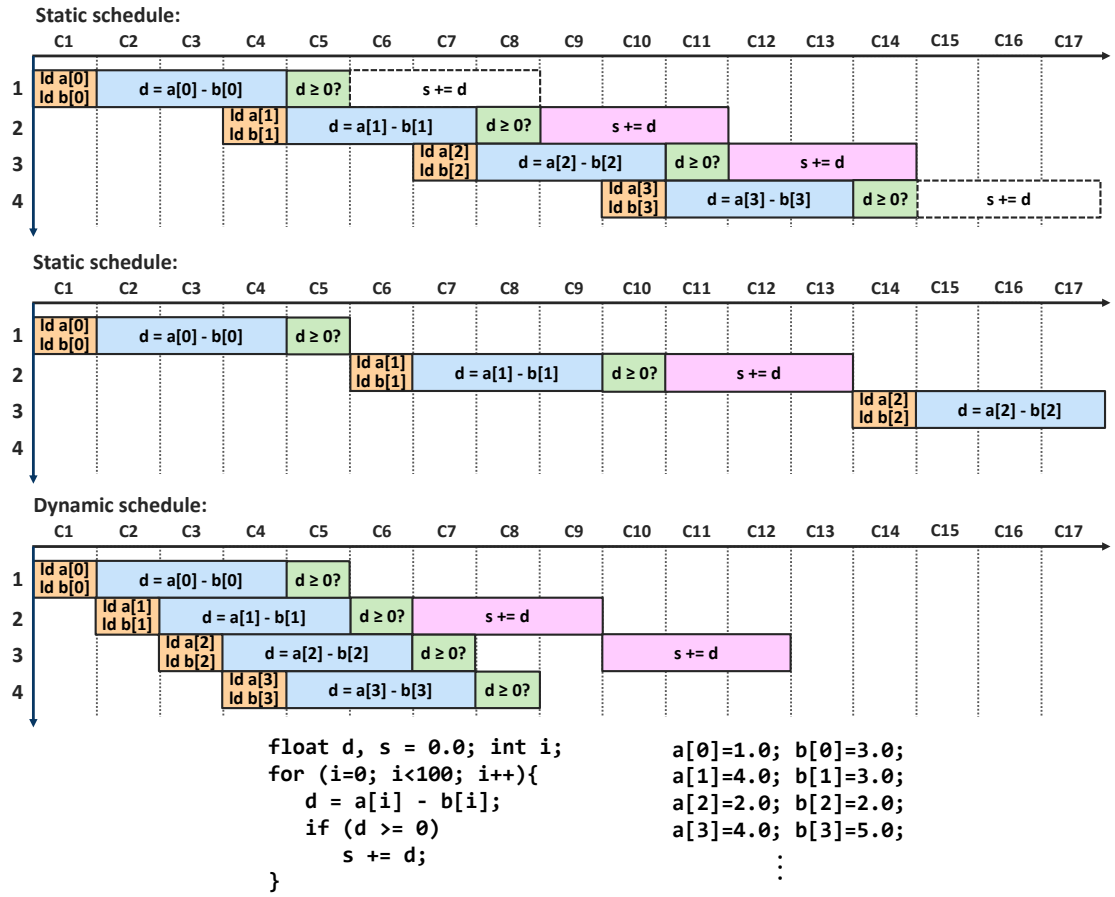


Figure 2.3 – Static vs. dynamic scheduling. The top and middle schedule (realized as a pipeline and a sequential state machine, respectively) are possible with standard HLS tools. A dynamic schedule (bottom) is achievable with our approach; it results in the best possible parallelism which is reduced only in the presence of actual loop-carried dependences.

a new loop iteration starts on every clock cycle and the resource requirements are minimal (i.e., new data is inserted into the single adder and the single multiplier on every cycle).

Loop pipelining is one of the key performance optimizations in HLS—as the example above suggests, it allows loop iterations to overlap in the best possible manner while honoring all data and control dependences of the program. The technique originates from software pipelining techniques for VLIWs, which rely on modulo scheduling algorithms to exploit instruction-level parallelism among successive loop iterations [82, 103]. A pipeline is characterized by its *initiation interval* (II), i.e, the number of clock cycles between consecutive loop iterations; the ideal II is equal to 1, as is the case for the pipelined schedule in the figure.

2.2 Why Dynamic Scheduling?

To illustrate the need to incorporate dynamic behavior into HLS, we revisit in Figure 2.3 the example from Figure 1.1b.

As described before, this loop has a control flow decision (`if`) which depends on the actual data being read from arrays `a[]` and `b[]`. Whenever the condition is true, a long-latency operation (`s += d`) introduces a dependence between iterations. A typical HLS tool needs to conservatively reserve a "space" in the schedule for this operation, even when it does not occur, as shown on the top of Figure 2.3. An alternative could be to avoid pipelining the loop and creating a sequential finite-state machine to achieve the middle schedule in Figure 2.3, where indeed cycles are spent for the addition only when needed; however, the decision of not pipelining the loop has removed one of the foremost potentials for parallelism (in this case, the memory reads, the subtraction, and the comparison are perfectly independent across iterations and could be pipelined).

Obviously, a good schedule is the bottom one in Figure 2.3: the operations of different iterations are overlapped as much as possible and the parallelism is reduced only when the dependence is actually there (that is, when the addition is executed). Such behavior is beyond what a statically scheduled HLS tool can achieve. In recent years, many authors explored workarounds for such problems—we will discuss these efforts in Chapter 8—but dynamically scheduled circuits represent the most general solution to the problem.

2.2.1 Dataflow Circuits

Dataflow or latency-insensitive circuits [19, 37, 116, 45] realize the dynamic behavior in Figure 2.3. These circuits are built out of units which communicate using pairs of handshake control signals; data is propagated from unit to unit as soon as memory and control dependences allow it and, otherwise, stalled by the handshaking mechanism.

Figure 2.4 shows a simplified version of a dataflow circuit implementing the loop of Figure 2.3. Besides normal datapath units, this circuit uses a few control units labeled buff, merge, select, fork, and branch. All directed edges in the figure represent data signals accompanied by handshake control signals. The loop to the left of the figure shows the part of the circuit which updates the iterator `i`: At the beginning, the constant 0 is sent from the entry point. The merge unit takes this value and passes it further. The buffer unit is the register which holds `i` and distributes it on the next clock cycle to three consumers through the fork unit; all successors must consume the value before the fork accepts a new input value. The left branch compares the incremented `i` with the loop bound; if the bound is not reached, the new value of `i` is sent back to the register by the branch unit through the merge. Meanwhile, the other outputs of the first fork use `i` to access `a[]` and `b[]` and to compute the subtraction, which is propagated to the rest of the circuit.

The key to a good execution of this loop is that, ideally, a new value of `i` should be used to start computing `a[i] - b[i]` on every cycle. This is the case in this circuit, contrary to a conservative statically scheduled one: The cycle on the left of Figure 2.4 is completely combinational excluding the buffer and thus a new value for `i` can be computed on every cycle. It is the right part of the circuit which can delay this: when the `if` is not taken, the result of the

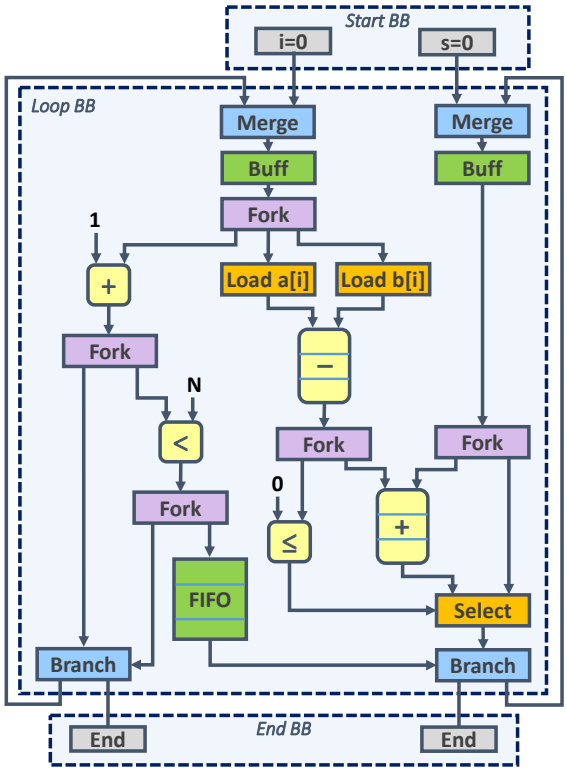


Figure 2.4 – A dynamically scheduled, dataflow circuit. This circuit implements the code from Figure 2.3 and achieves the optimal execution schedule (bottom schedule in that figure).

addition is dumped by the select unit as soon as it arrives through the merge and the old value of `s` becomes immediately the new value that is sent back to the adder on the following cycle; if, on the other hand, the result is needed, the select will wait for the sum to complete and the adder will be stalled next cycle waiting for its right operand. Ultimately, a new subtraction will not be computed and the memory accesses will not be performed due to backpressure from the adder; the top fork will not allow a new `i` to proceed on the right branch. This slows down the initiation of the loop and is exactly what the dynamic schedule in Figure 2.3 shows.

2.3 Synthesizing Dataflow Circuits

In this section, we detail our methodology to convert an arbitrary piece of code into a dataflow circuit. We first present the dataflow units we use; we then show how we implement control flow, ensure deterministic behavior, and construct the datapaths of our circuits.

2.3.1 Dataflow Units

Dataflow circuits are built out of *units* that implement latency-insensitivity by communicating with their predecessors and successors through *channels* composed of data lines and paired with handshake control signals: a *valid* signal indicates that a unit is sending a valid piece of data,

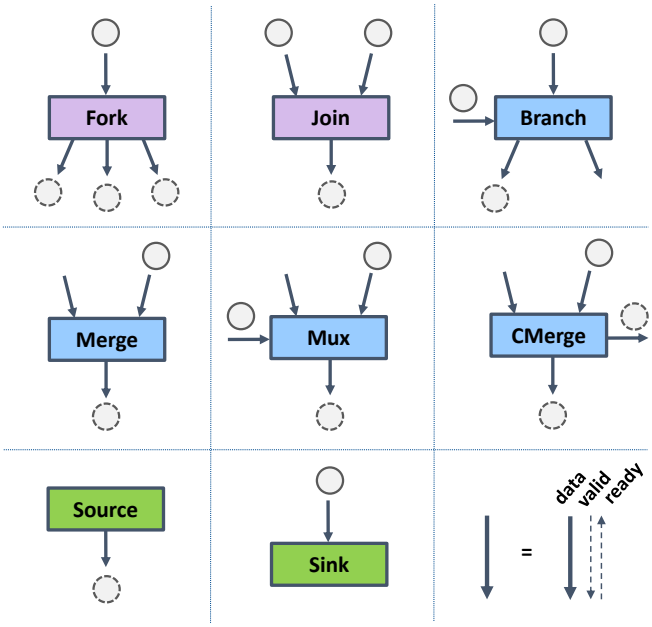


Figure 2.5 – Dataflow units. All data channels are paired with bidirectional control signals, which indicate the validity of data and the readiness of the successor unit to accept it.

referred to as a *token* [90], to its successor(s), whereas the *ready* signal informs the predecessor(s) that a unit can accept a new piece of data.

Figure 2.5 outlines the dataflow units we use; their gate-level descriptions can be found in literature [67, 37].

- An *eager fork (fork)* replicates every token received at the input to multiple outputs; as soon as one successor is ready to accept the token, the fork sends it to the successor; however, the fork can accept a new token only after all successors have accepted the previous one.
- A *lazy fork (lfork)* has the same functionality as the eager fork; however, it distributes a token to all successors at once (i.e., all successors must be ready for the lazy fork to send the token).
- A *join* acts as a synchronizer—its output is triggered only after all of its inputs become available.
- A *branch* implements program control-flow statements; it dispatches a token received at its single input to one of its multiple outputs based on a condition.
- A *merge* is a nondeterministic unit which propagates a token received on any of its input to its single output.
- A *mux* is a deterministic version of the merge; it propagates to its single output the input token selected by a control input.
- A *control merge (cmerge)* is a merge which, apart from the data output, has an output which indicates which of the inputs was taken by the merge.
- A *source* can always issue a valid token to its single successor (e.g., a constant).

- A *sink* is always ready to consume tokens from its single predecessor; the token is simply discarded in the sink.

In addition, we use any functional unit the code requires, such as integer and floating-point units. Units that require multiple operands contain a join to trigger the operation only when all inputs are available. Our circuits will require *buffers* which serve as registers in standard synchronous designs—we will discuss their properties and placement in Chapter 3. Finally, our circuits will interface to memory using read and write ports, yet, interfacing to memory is challenging due to the out-of-order nature of our system; we will address this issue in Chapter 5.

2.3.2 Implementing Control Flow

The starting point for out circuit generation is a standard compiler intermediate representation in *static single assignment* (SSA) form, where every variable is defined only once; *phi* nodes are used to define a variable from multiple definitions along multiple control paths [110]. The *control-flow graph* (CFG) of a program is organized into *basic blocks* (BBs), i.e., straight pieces of code separated by control flow decisions. Each BB contains a dataflow graph of program instructions; it receives *live-in* variables from the predecessor BBs and produces *live-out* variables for the successor BBs. Transforming the DFG of each BB into a corresponding interconnect of dataflow units is relatively straightforward—we will describe this process in Section 2.3.4—but there are a couple of problems when implementing control flow and sending values from one BB to another due to the fundamental difference between software programs and dataflow circuits.

Figure 2.6 shows two examples: (1) In the example in Figure 2.6a, the variable `a` is defined in BB0 and used in BB2. A typical representation in a compiler (left of the figure) propagates the desired information directly from the source to the destination block (i.e., a live-in of a basic block comes from a basic block which is not its immediate predecessor). This flow does not pose problems in software, as successive values of `a` would be stored in a register of a processor or in memory and the last value used when BB2 is reached. (2) In the example in Figure 2.6b, BB1 is the only BB in the body of a loop and uses a value `a` produced in BB0. The value of `a` does not change during the execution of BB1 and is used at every execution of BB1. Again, the representation on the left would cause no problem in a processor—the value would be stored in a register or memory and read as many times as needed.

Directly implementing such connections in a dataflow circuit would result in incorrect behavior because every generated value is associated with a token; the number of tokens must exactly match the number of distinct uses. The cases in the left of Figures 2.6a and 2.6b violate this principle if implemented literally: (1) In the first case, if the control flow were {BB0-BB1-BB0-BB1-BB2}, two new values (with the respective tokens) for `a` would have been generated and sent to BB2; yet, BB2 can take only a single token and requires only the most recent value. The execution would be incorrect or the circuit would not terminate because the tokens not absorbed by BB2 would create backpressure to BB0 and stop it indefinitely. (2) In the second case, BB1 would not be able to execute repeatedly due to a starving input. Assuming the control

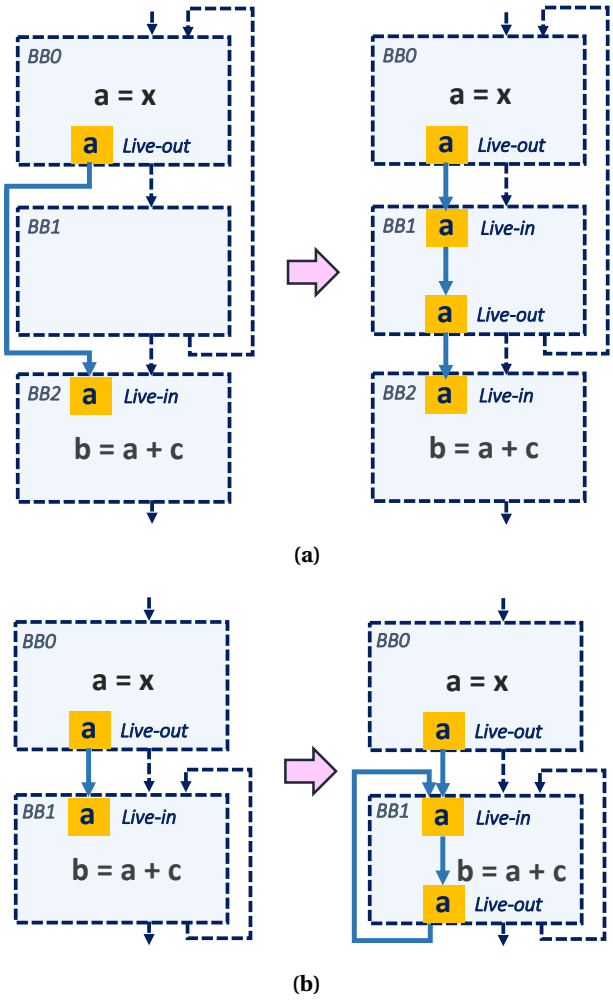


Figure 2.6 – Implementing control flow. The left circuits of Figures 2.6a and 2.6b show two cases where a direct conversion of a data and control flow graph into a dataflow circuit would fail. Coupling data and control to ensure correct token transfers between BBs is shown on the right of the figures: data is propagated exclusively from each BB to its immediate successors. For each live-in and live-out (shaded yellow in the figure), we employ merge and branch units, respectively.

flow is {BB0-BB1-BB1}, the first execution of BB1 will consume the single data token for `a` and any further execution of BB1 would stall indefinitely waiting for a token.

The solution to both problems is to strictly couple data propagation with control flow, as shown on the right of Figures 2.6a and 2.6b. The following properties must hold: (1) every BB must provide a live-out for every live-in of all of its immediate successor BBs and exclusively to them, and (2) every BB must receive all of its live-ins from its immediate predecessor BBs and exclusively from them. We implement these rules as outlined in Algorithm 2.1: (1) we employ a standard liveness analysis algorithm [110] to determine the live-ins and live-outs of each BB, (2) for every BB live-in and live-out, we instantiate a merge and a branch unit in the BB, respectively, (3) we connect all operations within a BB that use a live-in to the appropriate merge

```
// Input:  CFG (control-flow graph)
// Input:  DFG (SSA-based dataflow graph)
// Output:  DFG (dataflow graph with coupled data propagation
// and control flow)

// Determine live-ins and live-outs of each BB in CFG
liveIns, liveOuts = LivenessAnalysis (CFG)

// Place merge for every live-in in every BB
foreach bb ∈ CFG do
    foreach li ∈ liveIns (bb) do
        mg = CreateMerge(bb, li, DFG)
        // Connect all operations within the BB
        // that use the live-in to the corresponding merge
        foreach op ∈ operations (bb) do
            if li ∈ predecessors (op) then
                Connect (op, mg)

// Place branch for every live-out in every BB
foreach bb ∈ CFG do
    foreach lo ∈ liveOuts (bb) do
        br = CreateBranch (bb, lo, DFG)
        // Connect branch to corresponding merges
        // in successor BBs
        foreach bb_succ ∈ successors (bb) do
            mg = FindMerge (lo, bb_succ)
            Connect (br, mg)
```

Algorithm 2.1: Implementing control flow.

of the same BB (i.e., the merge will inject tokens into the BB body to trigger the execution of its operations), and (4) we connect the outputs of all branches to the inputs of the corresponding merges in the immediate successor BBs. In Figure 2.6a, this strategy results in merges for `a` in BB1 and BB2 and branches for `a` in BB0 and BB1. In Figure 2.6a, BB0 has a branch for `a` whereas BB1 has a merge and a branch.

This strategy guarantees that every piece of data is sent correctly from BB to BB, following the control flow of the program. Note that each BB contains as many merge units as it has incoming variables and as many branch units as it has outgoing variables. Some merges correspond to SSA *phi* nodes—they propagate into the BB a value chosen from one of the distinct predecessor values (based on the control flow), whereas other merges propagate a single value (coming from different control flow directions) to honor the rules above. This is the case, for instance, for the merge for variable `a` in BB1 of Figure 2.6b. All branches of a BB share the same condition and send tokens to the same successor BB based on a control flow decision.

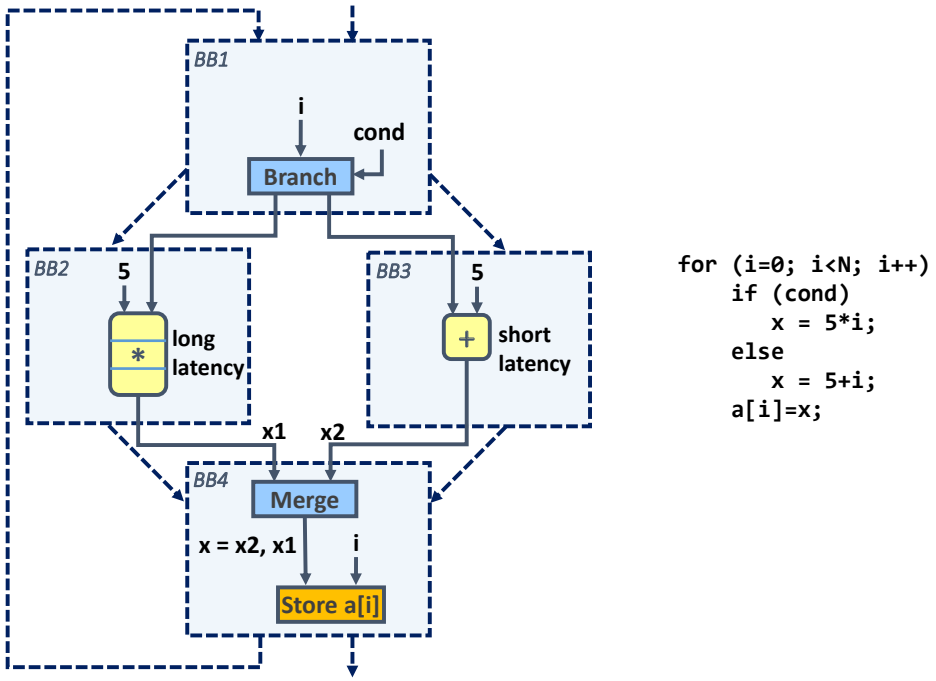


Figure 2.7 – Nondeterministic behavior at SSA *phis*. The token entering BB4 is produced either by BB2 or BB3; since these values are produced independently, the merge in BB4 may receive its inputs out of order.

2.3.3 Ensuring Determinism

Although different operations in a dataflow circuit may execute out of order, tokens are expected to arrive to each individual operator *strictly in order*. Yet, there is one particular case in which this property may not hold and which we discuss in this section.

The execution of our dataflow circuits is triggered by injecting a single token for each input (i.e., program argument) into the start BB. The tokens propagate through the BBs, following the control flow of the program—the BBs are triggered in exactly the same order as the software execution of the unmodified original program. When a single value propagates through the BBs, a token will always enter each BB from its single active predecessor—once the token enters through a merge, no other source can reinject a token into the merge until the merge itself produces a token, hence there is nothing that can interfere with the token ordering at the BB input. Tokens will never reorder inside a BB as it contains only straight and unconditional dataflow computation.

However, the situation is different in BB entry points where a value is redefined (i.e., when a token enters a BB through a merge which corresponds to SSA *phi* node)—as each input represents a distinct and, potentially, uncorrelated value, the input tokens may arrive in an order different than specified in the original program. An example of such a case is illustrated in Figure 2.7, which shows the CFG and a simplified datapath of the code in the figure. Assuming that the control flow is {BB1-BB2-BB4-BB1-BB3-BB4} (determined by the condition `cond` in BB1), the iterator from BB1 will first be sent to BB2 to compute the value of `x1`. This value takes multiple

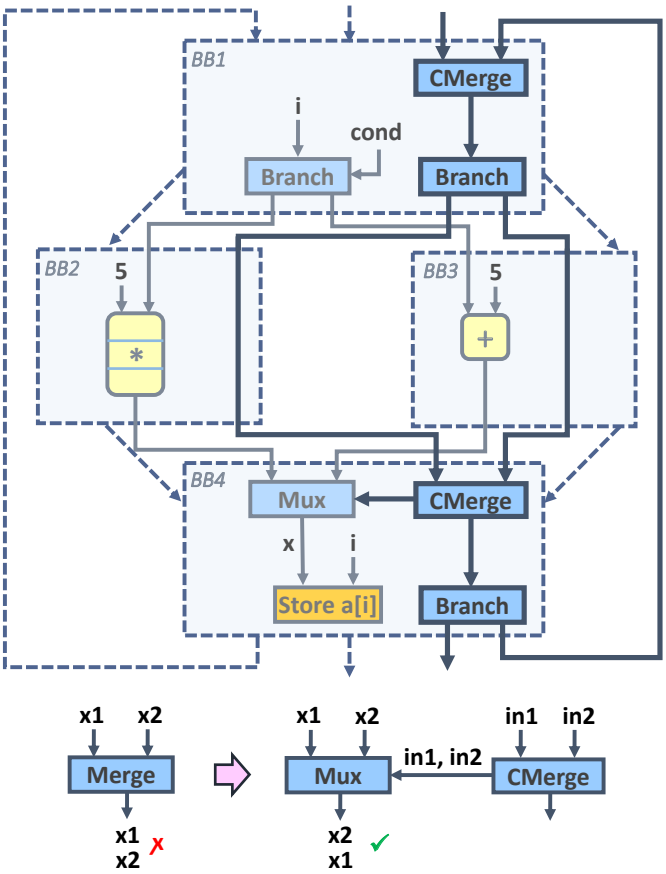


Figure 2.8 – Ensuring determinism. We extend the circuit from Figure 2.7 with a specialized in-order control network that follows the control flow of the program—the cmerges of this network communicate with the muxes of the same BB to indicate the correct input ordering.

cycles to compute, but the iterator can quickly propagate through BB4 and BB1 (the iterator path is omitted from the figure; as described in the previous section, it follows the control flow of the program). It will then enter BB3 which will trigger the short computation producing `x2`—this value may arrive to the merge in BB4 before the value of `x1`; the merge would send the values to the store out of order which would then produce incorrect results.

To ensure that tokens never enter a BB out of order, we replace every merge which corresponds to an SSA *phi* node with a mux unit described in Section 2.5. We generate an in-order control path that follows the control flow of the program through the BBs—essentially, a data-less variable which is a live-in and live-out of each and every BB. This path enters each BB through a cmerge, which connects to the muxes of the same BB and indicates the ordering of the inputs from which they will receive data. The extended circuit from Figure 2.7 is shown in Figure 2.8: in the previously discussed control flow sequence, the cmerge in BB4 would first receive a value on input `in1`, coming from BB2, and then on input `in2`, coming from BB3—it would indicate this ordering to the mux which would then not accept the value of `x2` before it has received the value of `x1`.

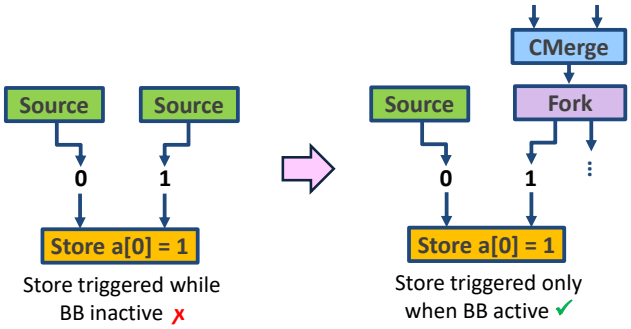


Figure 2.9 – Triggering constants. Setting constants as always valid (e.g., using a source) may incorrectly trigger operations (in this example, the store would constantly store data to memory, although its execution may not be determined by the control flow). In such cases, at least one constant should be connected to the in-order control network which ensures that the constant is triggered only when its BB is active.

This way of building dataflow circuits implies the following properties:

1. *Determinism.* The strict ordering of BBs reflected in the in-order control path guarantees that the execution is race-free.
2. *One token per cyclic path.* In the steady state of a CFG cycle (i.e., a loop) execution, each of its cyclic dataflow paths contains a single token (a token enters a cyclic path through a merge or a mux; the same token is sent back to the merge/mux as many times as the CFG cycle repeats before any other token can enter the cyclic path).
3. *Strict token ordering on a path.* If there are multiple tokens on an acyclic path, they could only be injected into it by repeatedly forking at every passage the single token of a cycle and the cyclic propagation of this token is determined by the in-order control flow decision.

2.3.4 Constructing the Datapath

Once the control flow is correctly handled, the BB internals are straightforward to design—each instruction is literally converted into its dataflow unit (i.e., a functional unit with inputs and outputs accompanied by handshake signals). When a unit has more than one direct successor, we place a fork to replicate the token into an individual token for each of the successors (i.e., for each point-to-point data transfer). Unused unit outputs (e.g., branch outputs without successors) connect to sinks which discard the unused tokens.

Some units (e.g., constants) do not have any inputs; we must ensure that they are appropriately triggered and executed. Keeping units without inputs always active (e.g., by setting a source as their input) may result in incorrect behavior, as they could trigger operations which are not supposed to execute. An example is shown in Figure 2.9: a store with a constant address and data would constantly send data to memory, regardless of the number of store executions specified by the program. Another case is that of a branch with a constant data input and a constant condition, which would constantly trigger a merge of some successor BB, even if this is not decided by the control flow. For this purpose, we exploit the in-order control network described in Section 2.3.3 and used to ensure determinism—we fork the token from this network and use

it to trigger operations with no inputs only and as many times as their BB becomes active, as shown on the right of Figure 2.9. Whenever a constant is an input to a unit that is triggered only when the BB is active (i.e., at least one of the unit predecessors is a live-in of the BB), the connection of the constant to the control path can be omitted and it can be triggered by a source instead, which reduces the complexity of the dataflow network. This is the case for the constants in BB2 and BB3 in Figure 2.7—the computational units will receive a data input and trigger the computation only if the corresponding BB becomes active, so both constants can be triggered with a source. Similarly, only one of the constant inputs to the store in Figure 2.9 requires a connection to the control network.

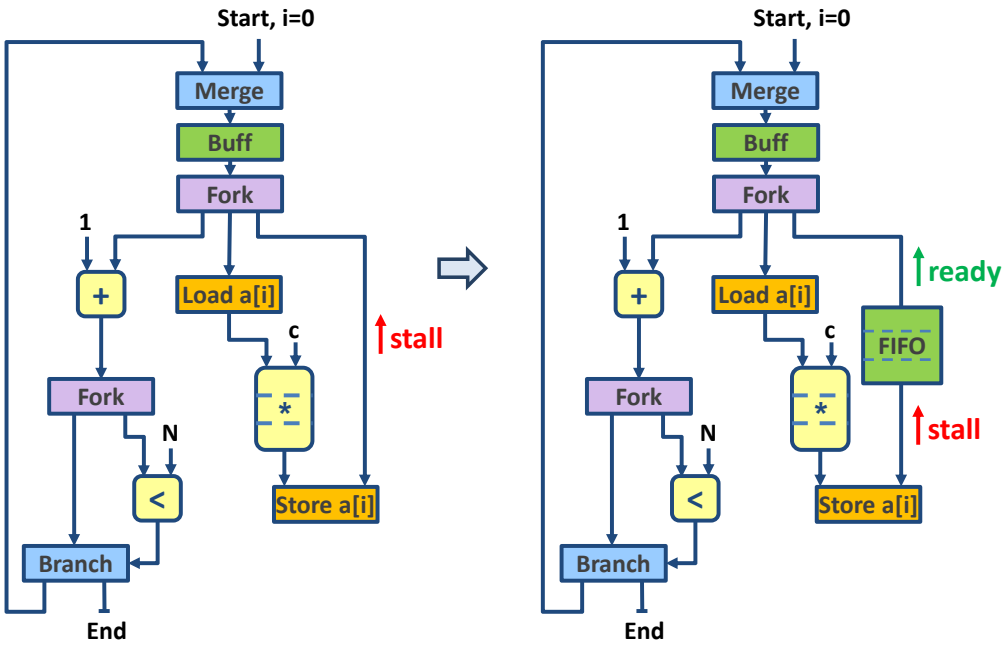
2.4 The Challenges of Dynamic Scheduling

In this chapter, we described a complete set of rules to synthesize C/C++ code into a functional (i.e., deadlock-free) dataflow circuit; we will present our fully automated HLS compiler which implements this HLS strategy in Chapter 9. However, the resulting circuits are not yet competitive with those produced by standard HLS tools: they are not able to achieve high-throughput pipelines in cases where standard tools can, nor are they able to save resources through sharing functional units. Although dataflow circuits have the ability to implement dynamic behaviors that superscalar processors regularly exploit (i.e., out-of-order memory accesses and speculation), the circuits described so far lack the necessary mechanisms to support these features. In the rest of this section, we illustrate these challenges and outline the most significant features and optimizations which we will discuss in the rest of this thesis.

2.4.1 Achieving High-Performance Pipelines

Dataflow circuits are naturally capable of pipelining, as the fine-grain handshake mechanism allows certain operations to run ahead and, consequently, enables executions of different operations to overlap. Yet, pipelining is not always possible due to *backpressure*: some paths take a longer time to consume a token and prevent potentially quick and independent paths from processing tokens at a high rate. This issue is illustrated on the left of Figure 2.10, showing the dataflow circuit implementing the code in the bottom of the figure: the fork could, in principle, issue tokens to the load (i.e., read memory port) on every cycle, but the path to the store (i.e., write memory port) stalls the first token until the multiplication completes, hence preventing new tokens from issuing to the load and limiting loop pipelining; the achieved schedule will, essentially, correspond to a nonpipelined schedule of a static HLS tool. Classic pipelining algorithms which standard tools exploit are not applicable in the absence of a static schedule; the solution here is to systematically identify and resolve backpressure to achieve the same pipelining effect.

Just like standard synchronous circuits, dataflow circuits require buffers, i.e., registers, which break combinational paths and, possibly, reduce the critical path of the circuit. Yet, in contrast to standard circuits, buffers can be placed on any channel without compromising the circuit




```
for (int i=0; i<N; i++){
   a[i] = a[i] * c;
}
```

Figure 2.10 – Adding buffers (i.e., FIFOs) to resolve backpressure and pipeline a dataflow circuit.

functionality. This property can be exploited to mitigate backpressure by inserting buffers into the paths that create stalls and lower system throughput, as illustrated on the right of Figure 2.10. Buffers used for regulating throughput typically have a larger capacity (i.e., they are implemented as FIFOs with multiple data slots) to hold all tokens issued by the predecessor before the successor is ready to accept them—in this example, the buffer requires 3 slots to constantly consume tokens from the fork; without the backpressure on the fork, the iterator loop can issue a new token on every cycle and achieve a perfect pipeline with an II equal to 1.

In Chapter 3, we present a performance optimization model [77] for dataflow circuits which relies on Petri net theory [90]. This model allows for resource-optimal buffer placement and sizing, with the purpose of maximizing throughput of the performance-critical loops at the desired clock frequency.

2.4.2 Saving Resources through Sharing

Standard HLS tools perform scheduling in conjunction with resource allocation and sharing [120, 34]; depending on the optimization objective, they trade off area and performance by deciding the cycle in which each operation executes and allocating units accordingly. The left of Figure 2.11 shows two possible schedules for the code in the figure. The first schedule is unconstrained in resources; by scheduling both multiplications in the same cycle, it employs two multipliers to achieve the ideal loop pipeline with an initiation interval of 1. The second schedule

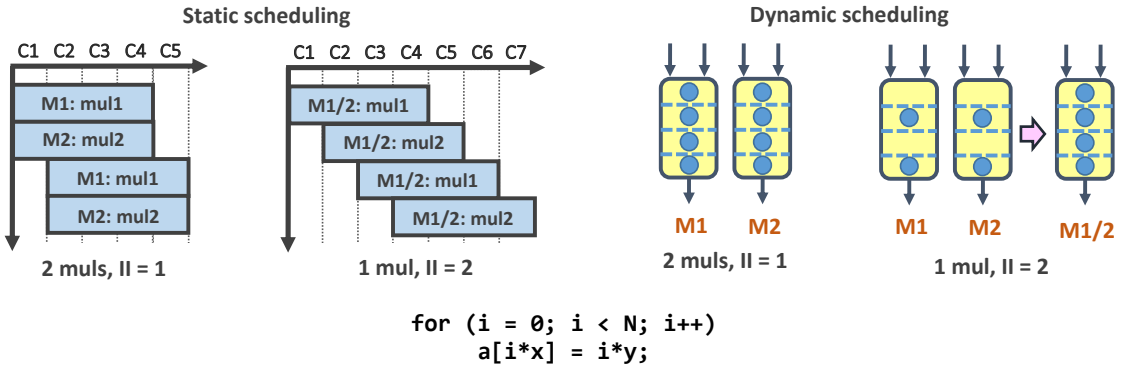


Figure 2.11 – Resource sharing in static and dynamic scheduling.

enforces a resource constraint of one multiplier; each multiplication must be scheduled on every second cycle and causes an increase of the II to 2.

Dataflow circuits face the same optimization objectives and area-performance trade-offs. Yet, in the absence of a predetermined schedule, it is challenging to determine which operations can share a functional unit without a performance penalty. Intuitively, one could rely on statistical information on unit utilization to decide what to share, as illustrated on the right of Figure 2.11: if the two multipliers are fully utilized (i.e., fully occupied with tokens), sharing would damage throughput; if they are only half-utilized, one could employ a single multiplier instead that would always be filled with tokens. Yet, this approach on its own may still compromise performance because the execution of some operations may be delayed with respect to their execution in the original circuit. More critically, although sharing seems to imply only some trivial circuitry, time-multiplexing units in dataflow circuits may cause deadlock by blocking certain data transfers and preventing operations from executing. Hence, a crucial step in making dataflow circuits resource-competitive with standard HLS is to systematically identify good sharing opportunities in an absence of a predetermined schedule, but also to build a sharing mechanism that always results in functional dataflow circuits.

In Chapter 4, we present a technique to automatically identify performance-acceptable resource sharing opportunities in dataflow circuits. Furthermore, we describe a sharing mechanism which achieves functionally correct and deadlock-free dataflow designs.

2.4.3 Introducing Out-of-Order Memory to HLS

One of the key enablers of dataflow pipelines lies in the ability to execute memory accesses in an order different than the one specified in the original program. As discussed in Section 1.2, pipelining in cases where memory dependences cannot be determined at compile time may be critical for dataflow computation to outperform statically scheduled HLS designs.

Out-of-order behavior has been exploited in out-of-order processors for decades [96, 104, 97]: load-store queues are used to ensure that all memory dependences are honored, while inde-

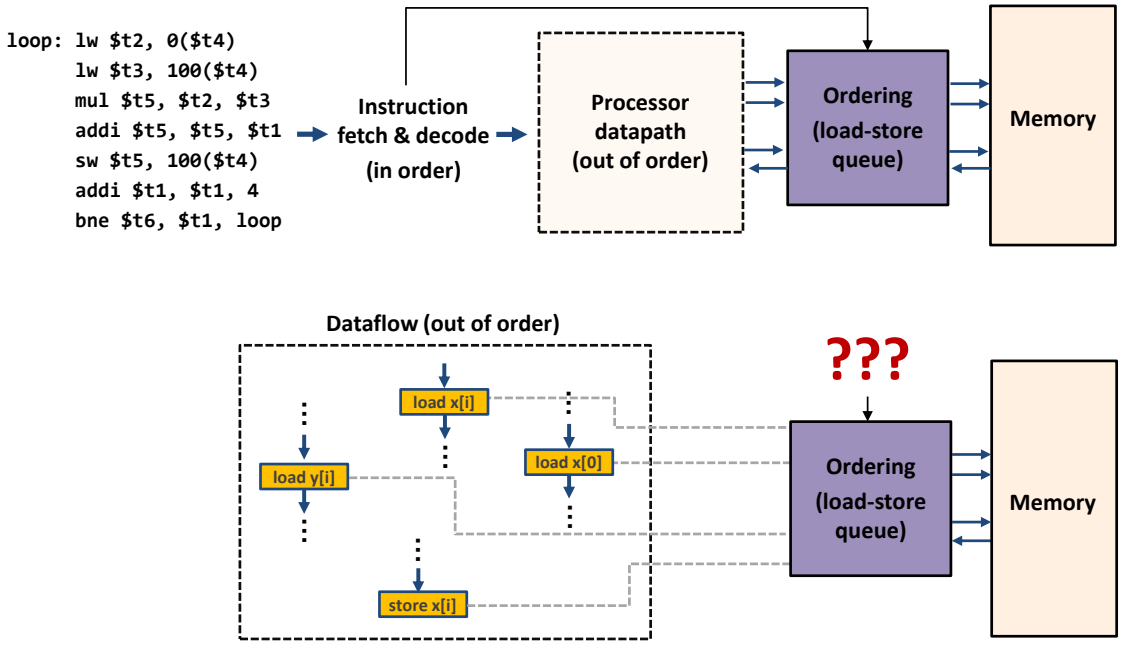


Figure 2.12 – Conveying program order to the memory interface. Program order is crucial to reorder memory accesses at the memory interface. In contrast to a processor, a dataflow system has no notion of fetching and decoding instructions to convey this order.

pendent memory requests may execute out of order for performance benefits. Dataflow circuits require the same functionality, but a processor LSQ cannot be employed directly because of a fundamental difference between the two systems, illustrated in Figure 2.12: In a processor, the notions of fetching and decoding instructions immediately convey the correct sequential order of requests at the memory interface. In contrast, dataflow circuits lack such notions and the information of the original sequential program order is lost; an LSQ receiving memory requests out of order would not be able to decide which reorderings are legal. Therefore, to employ an LSQ and truly benefit from out-of-order execution, dataflow circuits require an alternative way to perform allocation and to convey the correct order of memory requests to the LSQ.

In Chapter 5, we describe a practical way to organize an out-of-order memory interface (i.e., a load-store queue) for dynamically scheduled circuits so that it can correctly handle memory accesses arriving in arbitrary order while still respecting data dependences [71].

2.4.4 Minimizing the Complexity of the Memory Interface

Although LSQs enable dataflow circuits to achieve high performance in situations which static scheduling cannot efficiently handle, they imply high resource requirements as well as power and clock degradation when implemented on an FPGA. Hence, it is beneficial to leverage compiler analysis to simplify the memory interface whenever possible—whenever the compiler can disambiguate memory accesses, groups of accesses that cannot mutually conflict can use

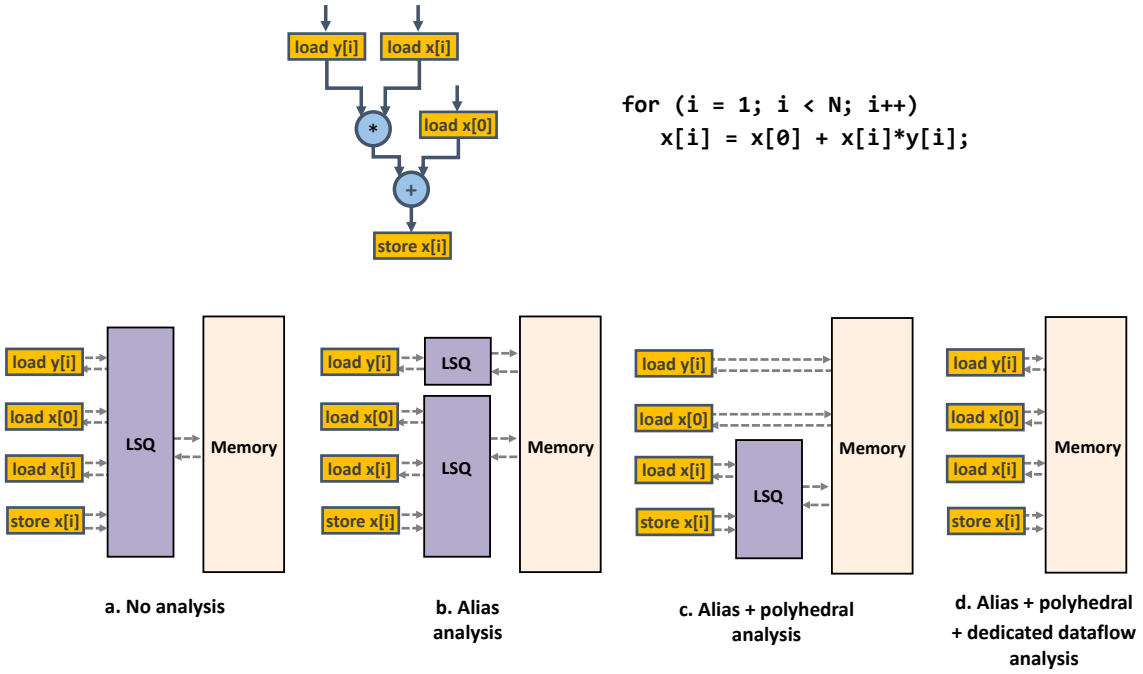
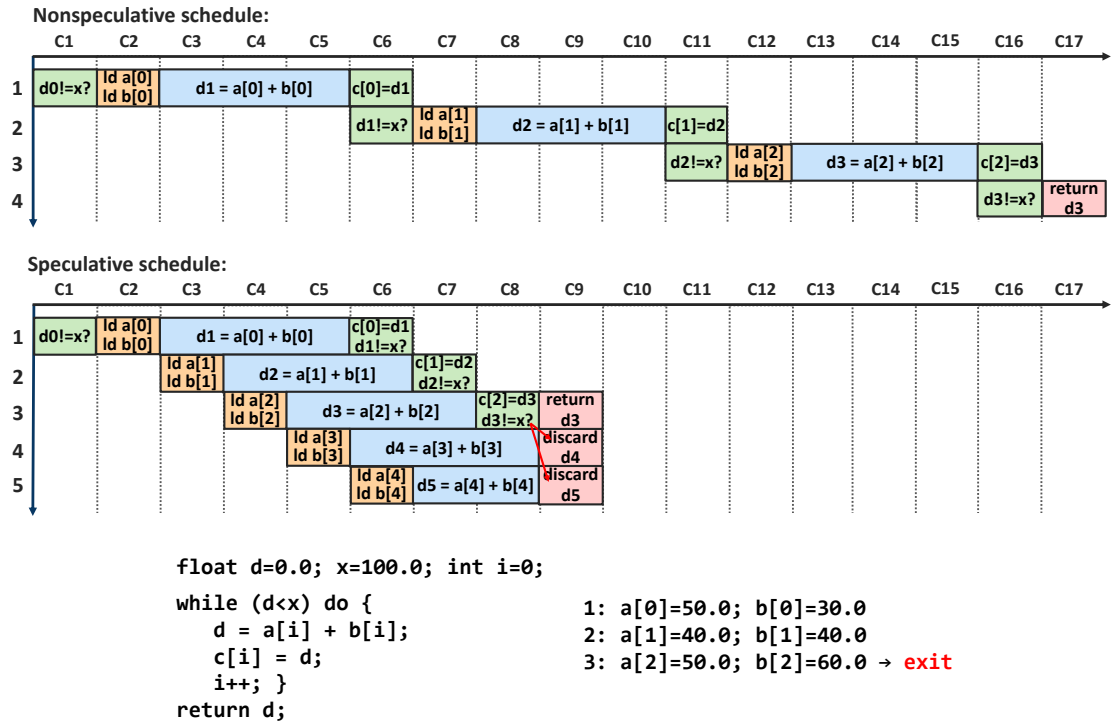


Figure 2.13 – Connecting a dataflow circuit to memory. The memory interface can be simplified (i.e., the number of accesses connected to memory using an LSQ can be reduced) by analyzing the memory accesses and excluding the presence of certain dependences.

separate LSQs, while those that certainly have no dependences with any other access can connect to simple memory interfaces. While standard HLS approaches analyze and optimize memory accesses, they are not always sufficient to obtain the best possible memory interface configuration in a dataflow system.

The code in Figure 2.13 shows a loop with multiple memory accesses which are analyzed and optimized using different memory analysis techniques. Without any memory analysis to reason about actual memory dependences, all accesses must connect to a single, large LSQ. By exploiting alias analysis and analyzing memory access patterns using polyhedral techniques, one can determine that some accesses (i.e., those accessing different arrays and those targeting different memory locations, respectively) cannot conflict; the memory interface can be simplified by employing multiple smaller LSQs and removing some LSQs altogether, as the second and the third memory configurations in the figure suggest. However, this is as far as standard techniques can optimize: the inability to reason about the order of executions of the load and the store to `x[i]` in an out-of-order dataflow system requires these accesses to connect to an LSQ. Yet, as it is immediately evident from the datapath in the figure, the two accesses naturally occur in the correct order as the load produces the data for the store—one can omit the LSQ completely without compromising correctness. Hence, to minimize the complexity of the memory interface, it is critical to determine cases where the temporal ordering of particular memory accesses is guaranteed by the presence of data dependences; this information cannot be captured with existing memory analysis techniques.




```
float d=0.0; x=100.0; int i=0;
while (d<x) do {
   d = a[i] + b[i];
   c[i] = d;
   i++; }
return d;
```

```
1: a[0]=50.0; b[0]=30.0
2: a[1]=40.0; b[1]=40.0
3: a[2]=50.0; b[2]=60.0 → exit
```

Figure 2.14 – Nonspeculative and speculative schedule for the code in the figure, repeating the situation from Figure 1.1c.

In Chapter 6, we explore techniques for reducing the cost of the memory interface in dataflow designs. Apart from exploiting standard memory analysis techniques, we present a specialized dataflow analysis [69] which relies on the topology of the control and dataflow graphs to infer memory order with the purpose of minimizing the LSQ size and complexity.

2.4.5 Enabling General Speculative Execution

As in computer architecture, dynamic scheduling paves the way to one of the most powerful ideas in computing: executing some operations before one has the certitude that they are actually needed or that it is correct to execute them. Speculation can significantly improve the execution of loops where the condition on the loop continuation that takes very long to compute by predicting very early whether it makes sense to execute *tentatively* another iteration. Similarly, speculation can further improve the problem of memory dependences, not only by reordering accesses once the lack of dependence is known but even by assuming independence early on and reverting back if the prediction was wrong.

The example in Figure 2.14 illustrates the need to enable speculative execution in HLS. A standard, nonspeculative schedule (repeated from Figure 1.1c) allows a new loop iteration to start only after the condition to exit the loop (which, in this example, takes multiple cycles to compute) has been checked; therefore, the loop cannot be pipelined. In contrast, a speculative

system would achieve the lower schedule which tentatively starts a new loop iteration on every clock cycle, before the loop condition is known.

The ability to implement speculation depends on reliable mechanisms to revert state changes due to wrongly executed operations and discard misspeculated values—in this example, the speculatively computed values from iterations 4 and 5 must be discarded and the result from the third iteration must be returned. In processors, this functionality is entrusted to reorder buffers and store queues [64], but these centralized units are not present in dataflow circuits. Therefore, the challenge in supporting general forms of speculation lies in the ability to create a distributed squash-and-replay mechanism in a generic dataflow network.

In Chapter 7, we describe a generic framework for handling speculation in HLS [74]. The idea is to trigger parts of the circuit to execute speculatively; special state-holding components hold the speculated data until the condition has been determined and dedicated components discard misspeculated data when required.

2.4.6 A Complete HLS Methodology

All the features presented in this section are critical to truly benefit from dataflow design. After we detail our strategies to achieve these features in Chapters 3 to 7, we discuss in Chapter 8 what others have done before us to enable dynamic behaviors in HLS. We present our complete HLS tool in Chapter 9; we compare our dynamic solutions with their statically scheduled counterparts and discuss the area and performance tradeoffs of these two design strategies. Finally, we conclude this thesis with an outline of possible future directions for dynamic scheduling in Chapter 10.

3 Buffer Placement and Sizing for High-Performance Dataflow Circuits

Dataflow circuits are naturally capable of overlapping loop iterations yet, as mentioned in Section 2.4.1, their pipelining abilities critically depend on the placement and sizing of buffers. In this chapter, we simultaneously tackle two aspects which are crucial for achieving high-performance circuits: constraining the critical path and maximizing throughput. We discuss the difficulties of performing such optimizations in the context of dataflow designs and present a performance optimization model based on marked graph theory which achieves maximum circuit parallelism at the desired clock frequency and with minimal resource cost. Our performance optimization model supports important HLS features such as pipelined computational units, units with variable latency and throughput, and if-conversion.

3.1 Buffers in Dataflow Circuits

The circuits produced by the compilation strategy described in the previous chapter do not contain any buffers. In this section, we discuss buffer properties and their importance in obtaining high-performance dataflow circuits.

3.1.1 Buffer Properties

Dataflow circuits require buffers which serve as registers in standard synchronous designs. A buffer can hold a *token* (i.e., valid data) or a *bubble* (i.e., invalid data)—each time a token moves forward, a bubble moves in the opposite direction, similar to electrons and holes in semiconductors [56]. Every cycle of our circuit will always contain *at most one token* (see Section 2.3.3), whereas bubbles can be freely allocated by adding buffers. Buffers are characterized with two properties: (1) *transparency*, which indicates whether a buffer adds sequential delay onto a path (a nontransparent buffer is used to break the combinational delay and implies a 1-cycle latency, whereas a transparent buffer is implemented as a pass-through element and does not increase

This chapter is based on the work published at the *28th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2020 [77].

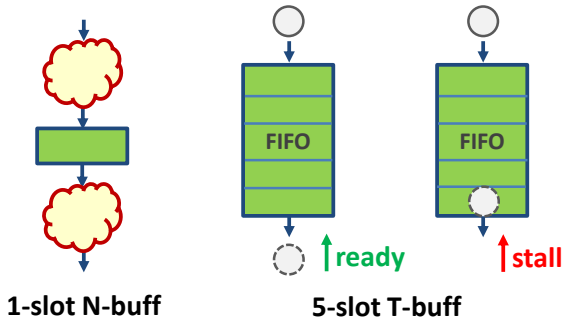


Figure 3.1 – Buffer properties. The figure contrasts a 1-slot nontransparent buffer, which can store a single token and always breaks the combinational path (thus incurring a 1-cycle latency), with a 5-slot transparent buffer, which can send data combinationally from input to output or store up to five tokens if the successor is not ready to consume them.

cycle count), and (2) *capacity* (i.e., number of slots), which is used to regulate throughput. These properties are illustrated in Figure 3.1: a single-slot nontransparent buffer is equivalent to a register in a standard synchronous circuit; a common FIFO of size N with a combinational path between input and output is here an N-slot transparent buffer.

3.1.2 Buffers and Circuit Functionality

Dataflow systems use distributed handshake signals to control the flow of data in the datapath. These signals implicitly take care of stalling early data items when they need to synchronize with late items [50]. Although buffers shift the executions of operations in time, their presence or absence does not affect the functional correctness of the system, as any consumer of multiple values synchronizes the corresponding valid tokens. Hence, contrary to registers in traditional synchronous designs, buffers can be placed on *any* channel of the dataflow circuit—due to its latency-insensitivity, this insertion will not compromise the functionality of the circuit [13], but may impact its timing and throughput.

3.1.3 Buffers and Avoiding Deadlock

The following conditions are necessary to ensure deadlock-free execution of dataflow systems: (1) Each combinational cycle must be broken with at least one nontransparent buffer; this requirement is analogous to that in standard synchronous circuits, where each combinational cycle needs to be broken using a register, and (2) each cycle in the circuit must contain at least one token and one bubble [38]; this requirement ensures that a token and a bubble can always exchange places and tokens can propagate through the cycle. As our circuit generation strategy guarantees that each cycle will have exactly one token, our combinational cycles will require at least two buffer slots to accommodate for the token and (at least) one bubble.

Figure 3.2 shows a combinational cycle of a dataflow circuit without a buffer, with a single-slot nontransparent buffer (satisfying the first requirement above), and with a two-slot nontranspar-

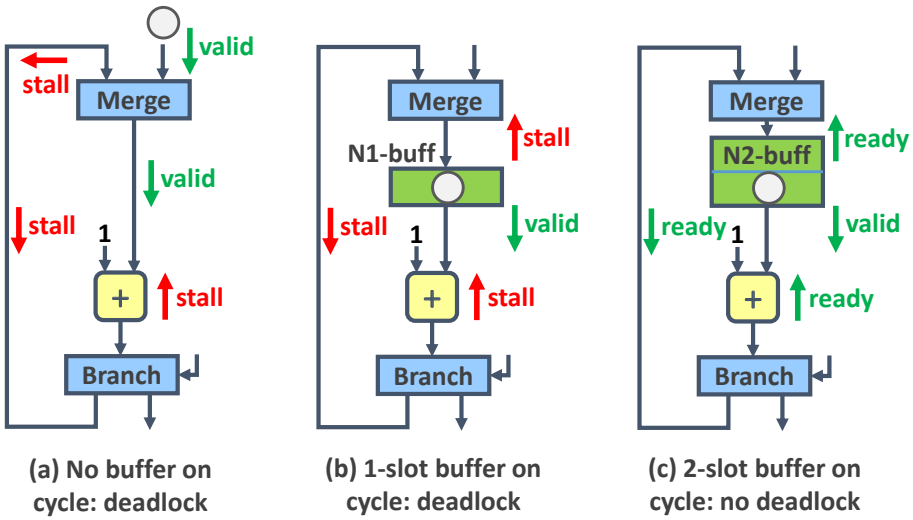


Figure 3.2 – Adding buffers. A combinational cycle without buffers or with a single buffer slot will cause deadlock, as the token will not be able to propagate through the cycle. At least two buffer slots are necessary to ensure deadlock-free execution.

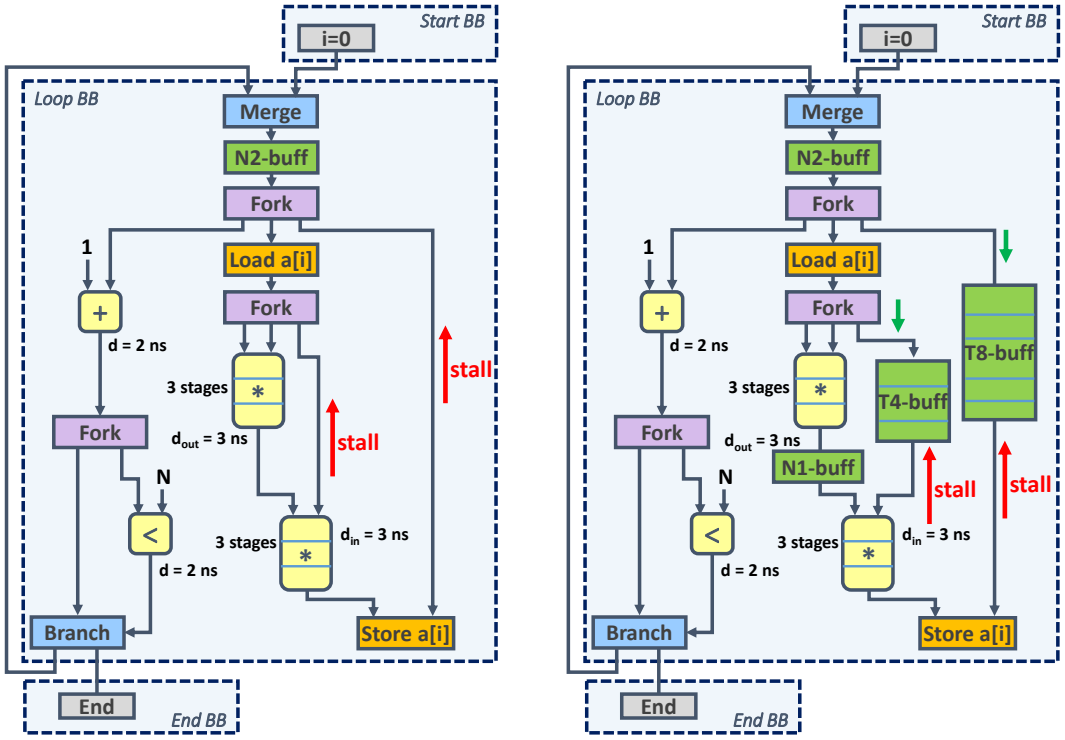
ent buffer (satisfying both of the requirements above). In the first case, a token at the input of the merge cannot propagate through the cycle due to the combinational relationship of the valid and ready handshake signals on the cycle. Adding a nontransparent buffer (N1-buff) breaks the combinational path and enables the token to enter the cycle, but there is no empty buffer slot (i.e., bubble) for the token to loop back through the merge. A 2-slot buffer (N2-buff) ensures deadlock-free execution as a token and a bubble can always exchange places.

3.1.4 Buffers and Performance

Figure 3.3a shows a dataflow circuit which calculates the cubes of array elements. The initial value of the iterator i is injected into the loop BB through a merge to trigger the computation start. The iterator is forked to a memory port to access an element of array a, which is sent to the pipelined multipliers to calculate the cube. The result is then stored back into the array. At the same time, the iterator value is incremented and compared to the loop bound. If the iterator has not reached the bound, its updated value is sent back through the branch to the merge, which triggers the start of the next loop iteration; otherwise, the program terminates.

The circuit in Figure 3.3a is completely functional and satisfies the correctness properties described in the previous sections, as the cycle of the iterator contains a buffer to break the combinational loop. However, this circuit fails to address two important performance aspects:

1. *Critical path.* The buffer is placed without any consideration for the combinational delays of the units (all non-zero delays are indicated in the figure) and therefore does not restrict the critical path in any way. The critical path of 6 ns is the sum of the output delay of the first pipelined multiplier with the input delay of the second multiplier.




```
int i = 0;
for (i = 0; i < N; i++)
   a[i]=a[i]*a[i]*a[i];
```

(c)

Figure 3.3 – Buffering for performance. A functionally correct, but unoptimized dataflow circuit (Figure 3.3a) implementing the code from Figure 3.3c contains a buffer placed to break all combinational loops. The optimized circuit (Figure 3.3b) has nontransparent buffers placed strategically to restrict the critical paths. The transparent buffers of larger capacity (i.e., FIFOs) in the paths with higher latency mitigate backpressure and allow achieving the ideal loop initiation interval (in this case, equal to 1).

2. *Throughput.* A major performance limitation is caused by backpressure, which we introduced in Section 2.4.1: some paths through the circuit take a longer time to process data and prevent the faster paths from consuming tokens at a higher rate. This effect may restrict loop pipelining—even if the need for another iteration can be decided very fast, new tokens may not be able to trigger the following loop operations because tokens from the previous iterations are stalled in the loop units.

In Figure 3.3a, the token carrying the array value a is forked into two pipelined multipliers, but the lower multiplier cannot accept the token until the upper multiplier is done computing (i.e., after 3 clock cycles). Similarly, the store can accept the iterator from the fork only after the two chained 3-stage multipliers produce a result. These stalls cause backpressure on their respective forks and prevent the short iterator path on the left from executing quickly: although

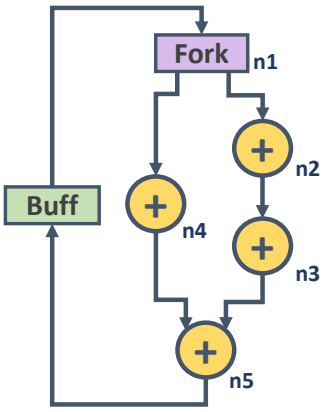


Figure 3.4 – A choice-free dataflow circuit, which has the properties of a marked graph. Circuits obtained out of high-level code (such as the ones in Figure 3.3) contain choices (i.e., control flow decisions through merges and branches) and cannot be represented as marked graphs.

a new iterator value could be computed on every clock cycle, the token with the updated value is stalled until the previous tokens have been consumed, which effectively lowers the initiation interval of the loop.

Figure 3.3b shows a circuit configuration with optimal throughput and the critical path constrained to 4 ns. The additional nontransparent buffer (N1-buff) lowers the critical path by breaking the combinational delay of 6 ns between the pipelined multipliers. Inserting transparent buffers of larger capacity into the slow paths corresponds to *slack matching* [87] and increases effective parallelism, as accumulating data in these buffers allows to trigger the faster paths at a higher rate. In this example, this is the case for the fast iterator path, which can now reenter the loop and trigger the start of a new loop iteration on every clock cycle, hence achieving a perfect pipeline with the initiation interval of 1.

3.2 Modeling Dataflow Circuits as Marked Graphs

In this section, we describe marked graphs, a particular class of Petri nets, which are the basis for the performance model we introduce in Section 3.3.

3.2.1 Marked Graphs

Marked graphs are a class of Petri nets [90] which represent concurrent behavior, but never have any *choices*, i.e., conditional execution. Figure 3.4 shows an example of a *choice-free* dataflow circuit which exhibits the properties of a marked graph. The buffer on the back edge of the circuit contains a token which infinitely loops through the combinational units: a token is forked from unit *n1* into both *n2* and *n4* concurrently, and the tokens from the two parallel paths are joined into a single token in *n5*—the transitions *n3* to *n5* and *n4* to *n5* always occur simultaneously.

This concurrency of marked graphs is the foundation of many linear algebraic techniques for their structural and performance analysis [102, 101, 16]; some address explicitly optimal buffer placement in choice-free dataflow graphs [13].

It is immediately clear that circuits such as the ones in Figure 3.3 do not exhibit the choice-free behavior of marked graphs, as each control flow edge between BBs represents a choice: the merge can accept the initial value of *i* from the starting point of the program, or the updated value sent back from the loop body; the branch can dispatch a value either through the back edge into the loop, or to the end point of the program, as determined by the branch condition.

3.2.2 Key Intuition

The performance of dataflow circuits critically depends on buffer placement and sizing, yet little is known about such optimizations. On the other hand, the timing properties of marked graphs have been extensively studied [13]. However, these techniques are not applicable in the context of dataflow circuits obtained from high-level code, which inevitably feature control flow and, therefore, cannot be represented as marked graphs. We here combine the knowledge from marked graph theory with dataflow circuits which implement choices in order to optimize their performance: our work is based on the observation that choice-free subgraphs with the properties of marked graphs can be extracted out of generic dataflow graphs. We describe an approach to perform this extraction and adapt an existing performance optimization model for marked graphs [13] to target dataflow circuits produced out of high-level code. We extend this model to support several typical HLS features, such as pipelined computational units and if-conversion. Finally, we discuss the optimization of complex dataflow circuits as well as methods for ensuring scalability of our approach. We evaluate our technique on a set of benchmarks obtained out of C code.

3.3 Optimizing Performance

In this section, we describe our strategy for extracting probabilistically most significant choice-free subgraphs of a dataflow circuit. We introduce our performance optimization model for obtaining the optimal buffer placement and sizes such that (1) the required cycle period is satisfied and (2) the throughput of the choice-free circuits is maximized. We begin with the single most important subgraph and then extend the approach to multiple subgraphs.

3.3.1 Extracting Choice-Free Dataflow Circuits

In this section, we describe our methodology for extracting the most significant *choice-free dataflow circuit* (CFDFC) from a dataflow circuit. We define a CFDFC as a dataflow circuit obtained from a cycle of the control-flow graph (CFG), i.e., from a CFG subgraph in which each BB has exactly one input and one output edge. A CFDFC is, therefore, (1) choice-free (i.e., the

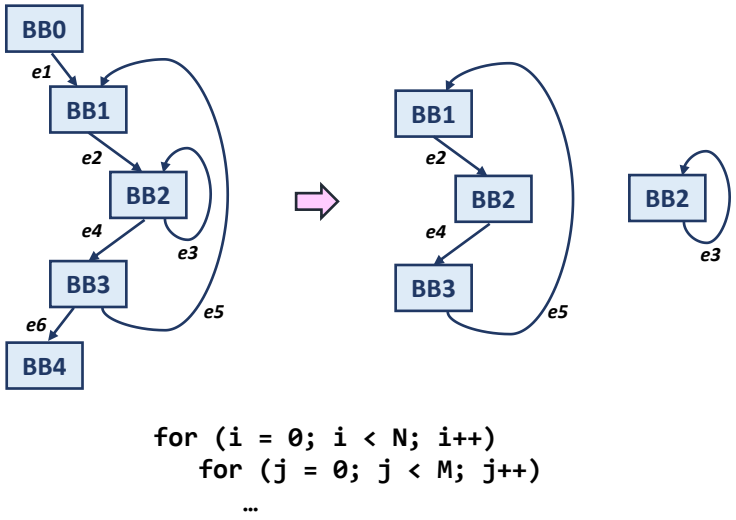


Figure 3.5 – Extracting CFG cycles. The leftmost graph is a control flow graph of a nested loop with two cycles. We optimize choice-free dataflow circuits (CFDFCs) which correspond to these cycles.

CFDFC has no control flow decisions), and (2) strongly connected (i.e., the CFDFC implements a loop of the program); hence, it can be represented as a marked graph. Figure 3.5 shows a control flow graph of a nested loop: it contains two cycles which, internally, correspond to two CFDFCs.

The performance optimization which we will introduce in Section 3.3.3 optimizes the most frequently executed CFDFC. We identify this CFDFC by finding the most frequently executed CFG cycle using an integer linear programming (ILP) model.

The ILP we employ has the following constants and variables:

- $\mathbf{N}_e$ (constant). Execution frequency of control flow edge e, i.e., the total number of times e executes.
- S_e^E (variable, binary). Indicates whether the control flow edge e belongs to the selected CFG cycle.
- S_b^{BB} (variable, binary). Indicates whether basic block b belongs to the selected CFG cycle.
- N (variable). Total number of times the CFG cycle executes.
- $\mathbf{N}_{\max}$ (constant). Upper bound on the number of executions, which has to be at least as large as the execution count of the most frequently executed edge of the CFG.

The following constraint states that the number of times the CFG cycle executes (N) corresponds to the minimum of the execution frequencies of the control-flow edges that belong to it. For evey edge e of the CFG,

$$N \leq S_e^E \cdot \mathbf{N}_e + (1 - S_e^E) \cdot \mathbf{N}_{\max}. \tag{3.1}$$

Here, $\mathbf{N}_{\max}$ ensures that N is not constrained by the execution frequencies of edges which do not belong to the loop.

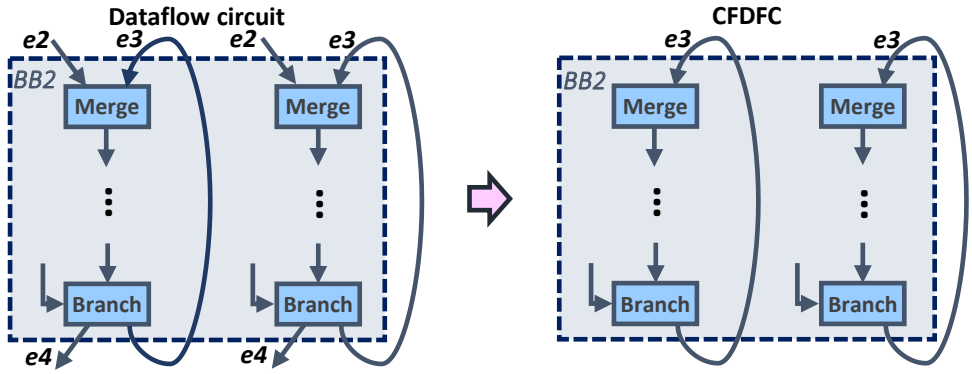


Figure 3.6 – Obtaining a choice-free dataflow circuit (CFDFC) from a dataflow circuit. A CFDFC contains all dataflow units and channels which belong to any of the BBs or edges of the extracted CFG cycle (in this example, BB2 and edge e_3 from Figure 3.5). Every merge and branch in a CFDFC have a single input and output edge, respectively.

If a BB is a part of the selected cyle, exactly one of its input and one of its output edges belongs to the cycle as well. For every BB b of the CFG,

$$S_b^{BB} = \sum_{e \in In(b)} S_e^E \tag{3.2}$$

and

$$S_b^{BB} = \sum_{e \in Out(b)} S_e^E. \tag{3.3}$$

Here, $In(b)$ and $Out(b)$ denote the sets of input and output edges of BB b, respectively. We assume that BBs at the beginning and end of the program have respectively no input and no output edge.

To ensure that only a single cycle is selected, only a single back edge of the CFG may belong to it:

$$\sum_{e \in Back(CFG)} S_e^E = 1. \tag{3.4}$$

Here, $Back(CFG)$ denotes the set of all back edges of the CFG. Back edges are typically defined as edges that point from a BB to another BB which dominates it (i.e., from a BB inside a loop to the loop header); they can be detected using classical dataflow analysis [1].

We formulate the cost function to obtain the most frequently executed CFG cycle as follows:

$$\max: \quad \sum_{e \in CFG} N \cdot S_e^E. \tag{3.5}$$

Once this cycle is identified, it is straightforward to find the corresponding CFDFC with its dataflow units and channels. The following properties hold for every unit of the CFDFC: (1) for every merge, only one input channel belongs to the CFDFC (corresponding to the chosen input

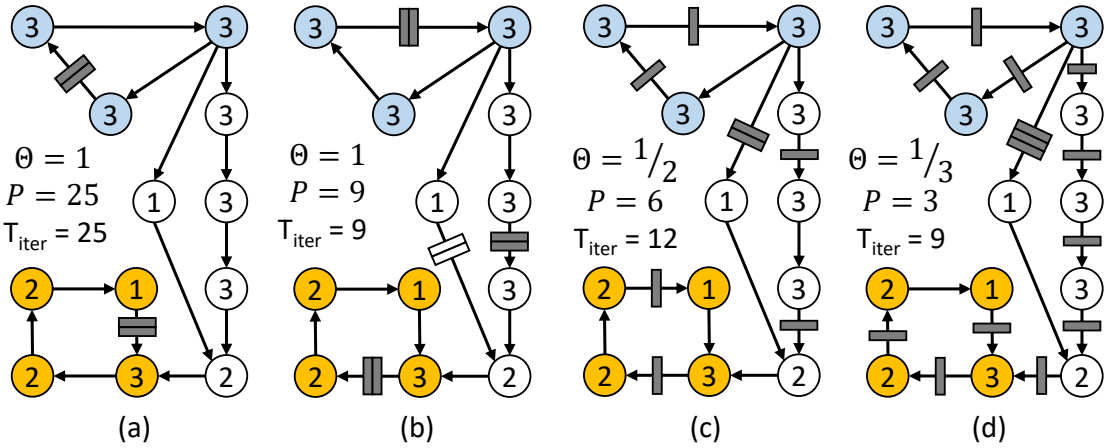


Figure 3.7 – Performance optimization of a choice-free dataflow circuit. Grey buffers are used for breaking combinational paths (i.e., nontransparent buffers). The white buffer is transparent and used for throughput optimization. The circuits in the figure differ exclusively in buffering, which directly affects their timing (i.e., the achievable clock period, throughput, and overall iteration time).

control flow edge of its BB), (2) for every branch, only one output channel belongs to the CFDFC (corresponding to the chosen output control flow edge of its BB), and (3) for all other units, all input and output channels belong to the CFDFC.

Figure 3.6 details the extraction of the most significant CFDFC of the program in Figure 3.5. The ILP will identify the self-loop of BB2 as the cycle with the highest execution frequency (i.e., the ILP result will be $S^{BB}_{BB2} = 1$, $S^{E}_{e3} = 1$, and all other BBs and edges are not selected). Therefore, for each merge in BB2, we keep only the input channel which originates from BB2 and belongs to edge *e3*; for each branch, we keep only the output channel leading back to BB2. All internal channels and units in BB2 belong to the CFDFC as well.

The approach that we have presented in this section will select one of the innermost loops of the circuit. We will extend our optimization model on multiple CFDFCs in Section 3.3.4.

3.3.2 Optimizing Choice-Free Circuits

The mathematical model presented in this chapter is based on the theory for performance analysis of concurrent systems inherited from timed Petri nets [102, 101, 16, 13]. We apply it to CFDFCs of the dataflow system, which can be represented as marked graphs (with functional units as nodes and channels as edges) to determine the optimal buffer placement and sizes.

The buffer configuration of a CFDFC determines its throughput Θ (i.e., the inverse of the initiation interval, 1/II): every cycle of the circuit has a cycle ratio defined as the inverse of the number of nontransparent buffers and the throughput is limited by the cycle with the minimum cycle ratio [102]. As every cycle contains at least a single nontransparent buffer, the throughput equals at most one (i.e., $\Theta \leq 1$).

Fig. 3.7 demonstrates the exploration space for the performance optimization of a choice-free dataflow circuit. In this example, there are two cycles that constrain the throughput. Every node (i.e., a functional unit of the dataflow circuit) is labeled with its combinational delay. Fig. 3.7(a) shows a solution with maximum throughput ($\Theta = 1$) by only putting 2-slot nontransparent buffers on the cycles, thus satisfying the requirement to accommodate a single token and a single bubble on each cycle. The cycle period $\mathbf{P}$ is then 25 and a cycle iteration takes 25 time units ($T_{iter} = P/\Theta$). Adding nontransparent buffers and moving the existing buffers reduces the critical path while maintaining the maximum throughput (Fig. 3.7(b)). To prevent the topmost loop to stall due to backpressure from the noncyclic path, an extra buffer (in white) has been added to one of the paths. Since it is not required to cut combinational paths, it can be implemented without adding any sequential delay (i.e., as a transparent buffer which acts as a FIFO, matching in size the nontransparent buffer on the right). Constraining the system to work with $\mathbf{P} \leq 8$ requires the addition of nontransparent buffers on the cycles, thus degrading the throughput. Fig. 3.7(c) shows a configuration with two buffers per cycle ($\Theta = 1/2$) and optimal period ($P = 6$) for this throughput, with $T_{iter} = 12$. Surprisingly, under the constraint $P \leq 8$, there is a more efficient configuration with lower throughput. The solution is shown in Fig. 3.7(d) with $\Theta = 1/3$ and $\mathbf{P} = 3$, resulting in 9 time units per iteration.

Solution (a) is the optimum in terms of area. Solution (b) is the optimum in terms of performance (T_{iter}) that minimizes area. Finally, solution (d) is the optimum in performance under the constraint $\mathbf{P} \leq 8$. The example shows the richness of solutions that can be explored in choice-free dataflow systems by changing exclusively the buffer positions and sizing—we will rely on this property to optimize the performance of our dataflow circuits.

3.3.3 MILP Model for Performance Optimization

We formulate our performance optimization model as a mixed-integer linear programming (MILP) model which determines the channels where buffers need to be placed as well as the buffer sizes. The model is based on the work of Bufistov et al. [13] for optimizing choice-free circuits—we here adapt it to generic dataflow graphs. We first present the model for a single CFDFC; in Section 3.3.4, we generalize our approach to multiple CFDFCs.

We class constants and variables of the MILP model into three groups: input constants (i.e., values given as input to the MILP), output variables (i.e., the solution of the buffer sizing problem), and internal variables (i.e., intermediate values found by the MILP solver but of little consequence to the user).

Input constants of the model.

- $\mathbf{P}$ (integer). Target clock period of the circuit.
- $\mathbf{P}_{\max}$ (integer). Upper bound on the clock period of the circuit, which has to be at least as large as any possible value of $\mathbf{P}$.
- $\mathbf{B}_c$ (binary). Indicates whether channel c is a back edge ($\mathbf{B}_c = 1$) of the dataflow graph.

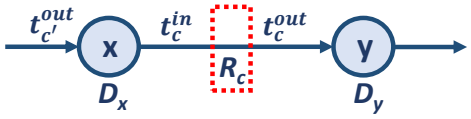


Figure 3.8 – Path constraints of the MILP model. These constraints ensure that the dataflow circuit meets the target clock period by accumulating delays across channels and inserting buffers (indicated with $R_c = 1$) to break the combinational path.

- $\mathbf{D}_u$ (real). Combinational delay of unit u.

Output variables of the model.

- R_c (binary). Indicates whether a sequential (nontransparent) buffer is present on channel c.
- N_c (integer). The number of slots of the buffer on channel c. The presence of a buffer implies at least one slot (i.e., $R_c \Rightarrow N_c > 0$). However, $N_c > 0$ and $R_c = 0$ indicates that a transparent buffer is present in the channel.

Internal variables of the model.

- Θ (real). Throughput of the CFDFC.
- $\overset{\bullet}{\Theta}_c$ (real). Average occupancy of channel c (token presence).
- $\overset{\circ}{\Theta}_c$ (real). Average emptiness of channel c (bubble presence).
- r_u (real). Fluid retiming of tokens across unit u.
- t_c^{in} (real). Arrival time at the the input of channel c (i.e., output of unit x, where $x \xrightarrow{c} y$).
- t_c^{out} (real). Arrival time at the output of the channel c (i.e, input of unit y, where $x \xrightarrow{c} y$).

We now describe the constraints of the MILP, grouped into path, throughput, and buffer sizing constraints.

Path constraints. These constraints ensure that the entire circuit meets the target clock period. They are therefore applied to the *complete dataflow graph.* For every channel c of the dataflow graph,

$$t_c^{out} \geq t_c^{in} - \mathbf{P}_{\max} \cdot R_c, \tag{3.6}$$

with $t_c^{out} \geq 0$. This constraint, depicted in Figure 3.8, propagates the combinational arrival time at each channel. In case of the presence of a buffer ($R_c = 1$), the right term is guaranteed to be negative and t_c^{out} becomes zero, essentially disabling the further accumulation of delays through this channel. The constraint requires an upper bound of the maximum cycle period ($\mathbf{P}_{\max}$).

The following constraints model the propagation delay of each unit u of the dataflow graph with a pair of input/output channels $x \xrightarrow{c_1} u \xrightarrow{c_2} y$:

$$\mathbf{P} \geq t_{c_2}^{in} \geq t_{c_1}^{out} + \mathbf{D}_u. \tag{3.7}$$

The leftmost constraint enforces all delays to meet the cycle period $\mathbf{P}$. For simplicity, we assume that channels and buffers have zero delays. Channel, buffer setup, and clock-to-q delays could be easily incorporated into the model by adding the corresponding constants.

Throughput constraints. Our circuit construction guarantees that there is a single token on each cyclic path of a CFDFC. We initially consider that this token is placed on the back edge—once the buffers are assigned to the edges of the system, the throughput constraints will distribute the token across the cycle edges accordingly. These constraints are only applied to the *choice-free circuit (CFDFC)* obtained using the methodology described in Section 3.3.1. For every channel $u \xrightarrow{c} v$ in the CFDFC,

$$\dot{\Theta}_c = \mathbf{B}_c + r_v - r_u \tag{3.8}$$

$$\Theta \leq \dot{\Theta}_c \,/\, R_c. \tag{3.9}$$

The first constraint is analogous to the equations of classical retiming [84]; in this case, the variables are real instead of integers (i.e., fluid retiming) and $\dot{\Theta}_c$ represents the average number of tokens in the channel at the steady state of the system. The second constraint indicates that the system throughput is determined by the channel with a minimum average number of tokens among all channels with a nontransparent buffer. This constraint can be easily linearized taking into account that R_c is binary and $\Theta \leq 1$:

$$\Theta \leq \dot{\Theta}_c - R_c + 1. \tag{3.10}$$

For $R_c = 1$ (i.e., the channel contains a nontransparent buffer), the throughput is limited by the channel occupancy (i.e., $\Theta \leq \dot{\Theta}_c$). Otherwise (i.e., for $R_c = 0$), the throughput Θ is not constrained by the channel since the largest possible throughput value is 1 (and the right side of the equation will be greater or equal to 1).

Figure 3.9 demonstrates fluid token retiming based on the throughput and path constraints. The path constraints determine the buffer placement to achieve the target period of $\mathbf{P} \leq 3$. The values next to the buffers represent the token occupancies $\dot{\Theta}_c$. They indicate that every channel of the upper loop with a buffer will contain a token every 1 out of 3 clock cycles, whereas the buffer in the bottom left channel will contain a token 2 out of 3 clock cycles. The values next to the units represent the retiming values r which indicate how much of the token must be retimed from the initial position (i.e., the middle channel) to achieve the average occupancies $\dot{\Theta}_c$. All values that are equal to zero are omitted from the figure.

Buffer sizing constraints. Buffer sizing is essential for avoiding backpressure. It corresponds to adding empty buffer slots, which do not affect circuit functionality. The average occupancy of tokens and bubbles will determine the number of buffer slots at every channel of the CFDFC:

$$N_c = \dot{\Theta}_c + \mathring{\Theta}_c\,. \tag{3.11}$$

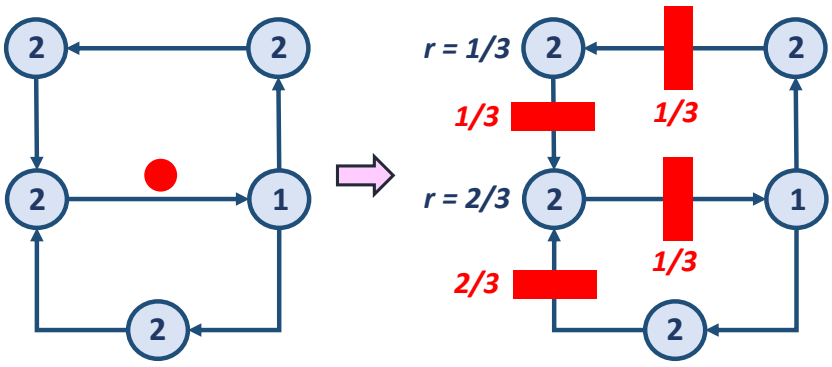


Figure 3.9 – Token retiming with throughput and path constraints for $\mathbf{P} \leq 3$. The token from the middle channel (left) is retimed during buffer assignment (right) and distributed in channels where the buffers are placed.

The constraint for bubble occupancy is dual to that of token occupancy (and can be linearized in the same manner); it ensures that each cycle has at least one bubble, thus avoiding deadlock:

$$\Theta \leq \mathring{\Theta}_c \,/\, R_c. \tag{3.12}$$

Cost function. Subject to the path and the throughput constraints, we maximize throughput Θ for a given clock period P, while accounting for the minimization of the total number of buffer slots in the channels of the dataflow circuit:

$$\max: \quad \Theta - \lambda \cdot \sum_c N_c, \tag{3.13}$$

where λ is a small coefficient that gives a lower priority to the minimization of buffer sizes. As already mentioned, the path constraints include the complete dataflow graph, whereas the throughput constraints apply to the most frequently executed CFDFC.

Without loss of generality, we here focus on maximizing the throughput of the system. The model that we have presented in this section could easily be employed with different cost functions and optimization objectives (e.g., minimizing the clock period or the buffer area cost under a throughput constraint [13]).

3.3.4 Optimizing Multiple CFDFCs

The model that we have presented so far optimizes only the single, most frequently executed CFDFC of the circuit. In this section, we extend our methodology to multiple CFDFCs.

We apply the ILP from Section 3.3.1 iteratively to extract one CFDFC after another based on their respective execution frequencies. After finding the most frequently executed CFG cycle, we update the execution frequencies by subtracting the execution values of the extracted CFG edges. Applying the ILP on the CFG while considering only the remaining execution values

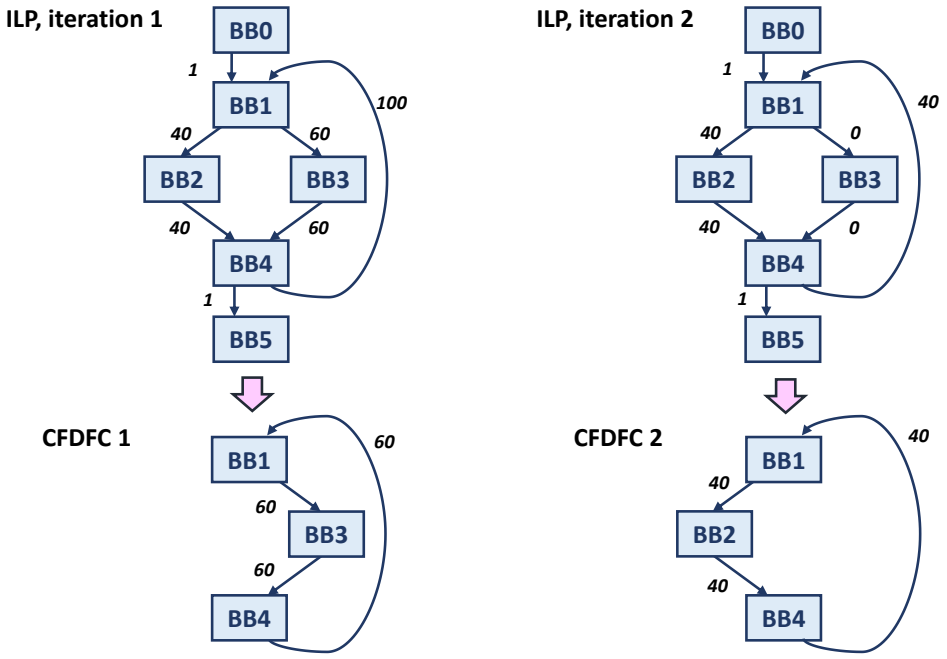


Figure 3.10 – Extracting multiple CFDFCs. The ILP from Section 3.3.1 can be iteratively applied while updating the execution frequencies of the CFG edges to extract one CFDFC after another. In the figure, the first extracted cycle (and the CFDFC which it represents) executes 60 times. After subtracting this value from the execution count of each corresponding CFG edge, we extract the next cycle of 40 iterations.

extracts the next cycle and its corresponding CFDFC based on its share in the runtime of the program. We illustrate this approach in Figure 3.10.

It is important to note that our ILP extracts cycles in the order of their importance (i.e., based on their fraction in the application runtime). We could also employ any algorithm for finding cycles in a directed graph [68], yet this approach would require extracting *all* graph cycles and subsequently sorting them based on their execution frequencies (by repeatedly identifying the most significant cycle and then updating the execution frequencies of all remaining cycles). The fact that our ILP simultaneously orders and extracts the cycles makes it possible to terminate the extraction as soon as appropriate criteria have been met (e.g., no remaining edge has an execution frequency above some threshold or the extracted cycles collectively represent a sufficient fraction of the application runtime). As we will discuss in Section 3.6, having such criteria is of great importance to limit the MILP runtime; moreover, the optimization of all CFDFCs is not always needed to maximize performance.

Optimizing multiple CFDFCs requires the extension of the MILP from Section 3.3.3 to maximize throughputs of all CFDFCs. For every additional CFDFC, the MILP includes an additional set of throughput and buffer sizing constraints (i.e., Equations 3.8 to 3.11). The cost function to maximize system throughput considers a weighted sum of the individual throughputs Θ of all extracted CFDFCs:

$$\max : \sum_i w_i \cdot \Theta_i - \lambda \cdot \sum_c N_c, \tag{3.14}$$

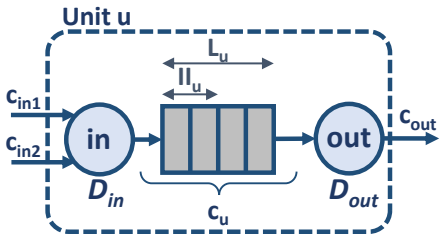


Figure 3.11 – A model of a sequential (pipelined) unit.

where the weight w_i of each throughput is proportional to the frequency of execution of each CFDFC (i.e., an approximation of the runtime fraction of each CFDFC in the program profile).

3.4 Modeling Computational Units and If-Conversion

The model that we have presented so far (as well as the model by Bufistov et al. [13] which our work is based on) only accounts for combinational nodes (i.e., combinational dataflow units). In this section, we extend the model to pipelined computational units and discuss how to apply it to units with variable II and latency. We then use these insights to model if-conversion, a typical HLS transformation.

3.4.1 Modeling Pipelined Units

To be able to handle cases such as the one in Figure 3.3, our MILP model needs to account for pipelined units. For this purpose, we characterize a pipelined unit u using two classic parameters: latency $\mathbf{L}_u$ and initiation interval $\mathbf{II}_u$. Figure 3.11 depicts our model.

The pipelined unit is modeled as two combinational units, separated by a channel c_u. The units *in* and *out* are represented with the unit input and output delay, $\mathbf{D}_{in}$ and $\mathbf{D}_{out}$. The channel c_u contains a nontransparent buffer with $\mathbf{L}_u$ slots, as the latency of the unit corresponds to the number of tokens the unit can hold. The delays $\mathbf{D}_{in}$ and $\mathbf{D}_{out}$ participate in the path constraints of the MILP (i.e., Equations 3.6 and 3.7), like those of any other unit. The maximal operating frequency of the unit, $f_{u,max}$, can be neither modified nor optimized by the MILP, hence we provide it directly as a constraint on the target clock period (i.e., $1/f_{u,max} \leq \mathbf{P}$). Unless this constraint cannot be met (i.e., the MILP cannot find a feasible solution for the given target period), it has no impact on the buffer placement and sizing.

The initiation interval of unit u puts a constraint on the average presence of tokens in channel c_u that cannot be greater than $\mathbf{L}_u/\mathbf{II}_u$. Thus, throughput constraints for channel c_u can be written as follows:

$$\dot{\Theta}_u = r_{out} - r_{in} \tag{3.15}$$

and

$$\Theta \cdot \mathbf{L}_u \leq \dot{\Theta}_u \leq \mathbf{L}_u/\mathbf{II}_u, \tag{3.16}$$

where r_{in} and r_{out} are the corresponding retiming variables for the input and output combinational units, *in* and *out*.

3.4.2 Modeling Variable Initiation Interval

The model from Section 3.4.1 assumes a constant latency $\mathbf{L}_u$ and a constant initiation interval $\mathbf{II}_u$ for each computational unit. Yet, this may not always be the case—we here consider units with a variable initiation interval, i.e., $\mathbf{II}_u = [\mathbf{II}_{u,min}, \mathbf{II}_{u,max}]$. Typical examples of units which exhibit such behavior are read and write ports which connect to memory through a load-store queue (LSQ)—we will discuss this unit in detail in Chapter 5. The role of the LSQ is to ensure that all memory accesses with dependences are executed in the correct order—based on the dependences present in the program, the LSQ may issue and receive data from the memory ports at different rates, hence lowering their effective initiation interval. For instance, if a load has a read-after-write dependence with a previous store, the LSQ will return its data only after the store completes; this effect will temporarily lower the rate with which the load port issues data into the circuit, hence resulting in an increased II. On the other hand, if there are no dependences, the load port can issue data at a high rate (i.e., with a low II, ideally equal to 1).

Like any other sequential unit, units with variable II require the formulation of Equation 3.16, which connects the unit II, $\mathbf{II}_u$, to the CFDFC throughput, i.e., $\Theta \leq 1/\mathbf{II}_u$. A higher $\mathbf{II}_u$ value results in a tighter throughput constraint; hence, modeling a unit with an $\mathbf{II}_u$ value which is larger than the II achieved during execution may conservatively constrain the throughput Θ. Consequently, the resulting buffer configuration may be suboptimal and the achieved circuit performance may be limited. The question here is, therefore, how to choose the appropriate value of $\mathbf{II}_u$ for Equation 3.16 when the II of a unit is variable.

Consider the circuit in Figure 3.12, with the timing parameters of each unit listed under (a); all units have fixed latencies, but the II of the multiplier varies between 1 and 2. If we include into the MILP model the higher II, $\mathbf{II}_{mul,max} = 2$, it will constrain the CFDFC throughput to 1/2 and the buffers will be sized accordingly—in this case, the capacity of the transparent buffer before the store will be set to 3. Although this buffer capacity is sufficient to sustain the throughput of 1/2, it will cause backpressure when the multiplier operates with a lower II, hence *always* (i.e., regardless of the actual II that the multiplier achieves) constraining the throughput to 1/2. In contrast, optimizing the circuit for $\mathbf{II}_{u,min} = 1$ (and, consequently, the throughput of 1) will result in a larger buffer capacity (here equal to 6)—the buffer will be fully utilized when the throughput is equal to one and underutilized otherwise, but it will never limit the throughput and damage performance.

Given that our goal is to maximize throughput (and, consequently, performance), we model each unit with its minimum initiation interval value, $\mathbf{II}_{u,min}$.

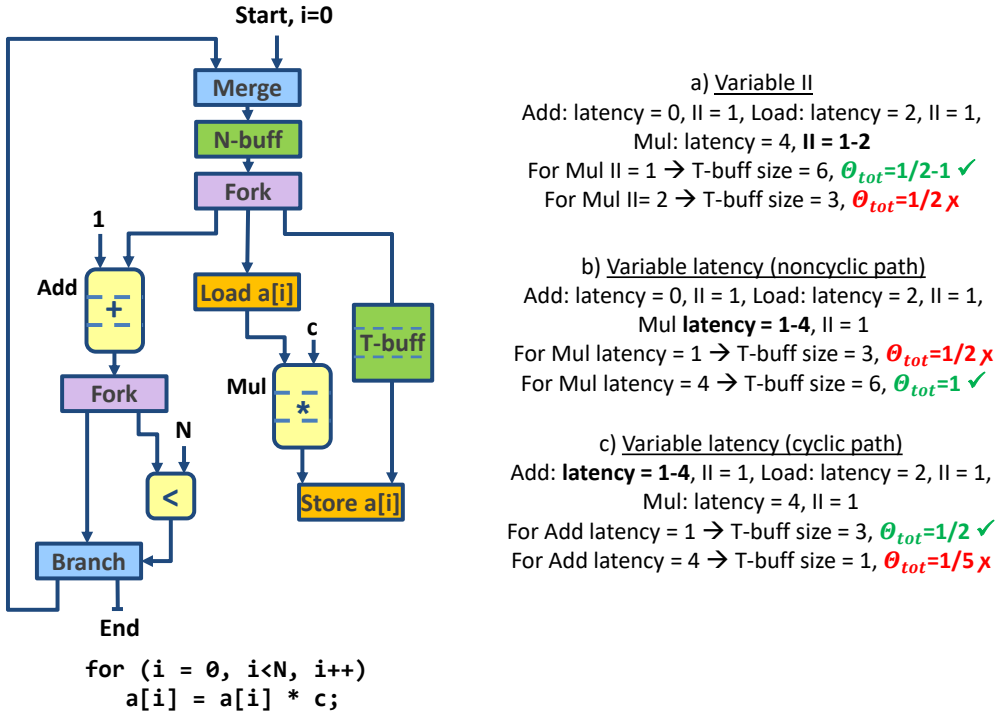


Figure 3.12 – Modeling variable II and variable latency. When a unit has variable II (case a), it is always desirable to model its best-case (i.e., lowest) throughput to achieve optimal buffer placement and, consequently, best possible performance. When a unit has variable latency (cases b and c), its model depends on whether the unit lies on a cyclic path; if so, it should be modeled with its minimum latency (so that it does not constrain throughput); otherwise, it is should be modeled with its maximum latency.

3.4.3 Modeling Variable Latency

Our circuits may also contain variable-latency units: for instance, computational units which can take a variable number of cycles to compute the result depending on the input data as well as load ports which wait a variable number of cycles for the memory to return the requested data. As in the case of variable II, the modeled latency value may have a significant impact on the circuit throughput and performance.

Consider the example in Figure 3.12, now with the parameters listed under (b), where now the multiplier latency ranges from 1 to 4 cycles. If the MILP model considers the minimum latency of 1, the capacity of the transparent buffer before the store will not suffice to eliminate backpressure when the multiplier latency is higher than 1. In contrast, if the model considers the maximum latency of 4, the MILP will produce a larger buffer which will relieve backpressure for every possible case, hence achieving the best possible performance.

The situation is different with the timing assumptions of bullet (c)—the variable-latency adder is on a cyclic path and its latency may constrain the CFDFC throughput. This effect occurs because there is always a single token on a cycle, as mentioned in Section 3.3.2; a long-latency unit on a cyclic path limits the rate with which the token can reenter the loop body, therefore

lowering the system throughput. Hence, considering the maximum latency value may create an overly-conservative throughput constraint and buffer placement, similar to the effect discussed in the previous section, i.e, the resulting buffers will not be able to sustain higher throughput, achievable when the unit latency is lower. Hence, when a unit is on a cyclic path of the CFDFC, it is desirable to consider its minimal latency; the MILP will optimize the system for the largest achievable throughput and place buffers accordingly.

Therefore, to maximize performance, we model the latency of a variable-latency unit $\mathbf{L}_u = [\mathbf{L}_{u,min}, \mathbf{L}_{u,max}]$ in Equation 3.16 as follows: (1) if a unit is on a cyclic path of a CFDFC, we consider its minimum latency, $\mathbf{L}_{u,min}$, and (2) if a unit is on a noncyclic path, we consider its maximum latency, $\mathbf{L}_{u,max}$.

It is important to note that a unit may be on a cyclic path through one CFDFC but on a noncyclic path of another (e.g., a loop-carried dependence of an innermost loop may not be on the cyclic path through the outer loop). As each CFDFC has its own set of throughput constraints, each CFDFC can consider the appropriate latency following the rules above to achieve the best possible throughput.

The rules presented in this and the previous section could easily be adapted to different cost functions and optimization objectives, discussed in Section 3.3.3. A possible modification would be to consider average latencies and initiation intervals; such an approach may result in lower resources (i.e., it may instantiate smaller buffers than the solutions we present here) but, in contrast to our approach, it would not guarantee optimal performance for every possible outcome.

3.4.4 Modeling If-Conversion

Compilers typically rely on optimizations such as if-conversion to convert conditional branches into predicated instructions [110]; this transformation usually requires a dedicated instruction which chooses one of the input values based on a condition (i.e., a select instruction). This instruction translates directly into the corresponding select dataflow unit; it can be implemented as a multiplexer which outputs data as soon as the condition and the chosen input are available, whereas other inputs are simply discarded upon (possibly late) arrival [22, 36].

The performance optimization model that we have discussed so far considers all operations within a BB as choice-free units—all inputs must become available for the unit to trigger. This model is not suitable for a select unit, which needs only one of its inputs (i.e., the input chosen by the condition) to trigger. If a select is on a cycle and one of its inputs takes more cycles to compute than the other, our model would assume the worst-case latency and conservatively model throughput, even if the long-latency input may actually be discarded and the computation can proceed early on. This is the case for the circuit in Figure 3.13: even if the multiplier input is not selected, the MILP accounts for the long-latency cyclic path (shown in red dashed) and limits the obtainable throughput, exactly as described in the previous section.

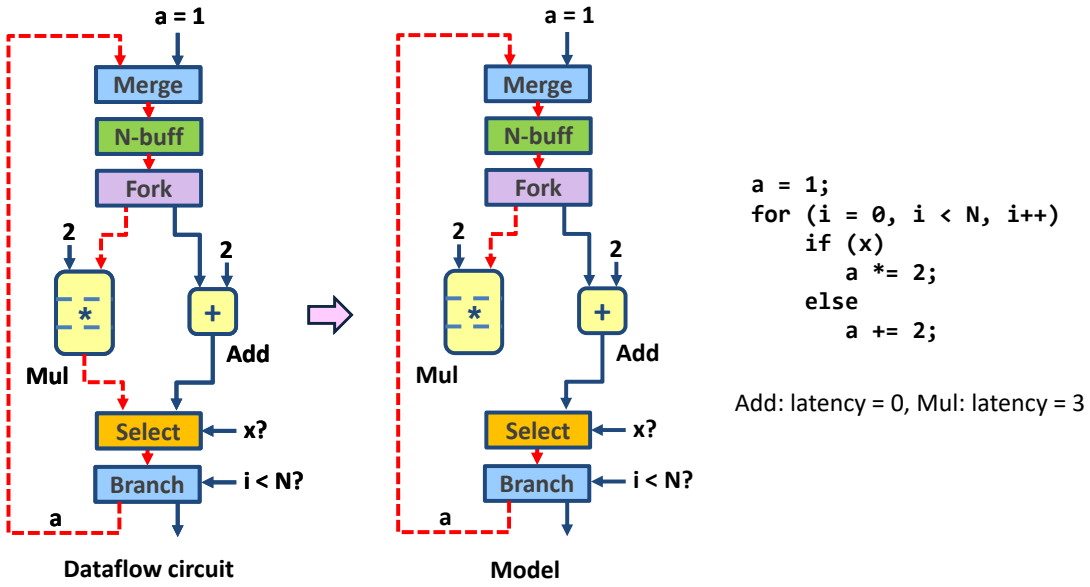


Figure 3.13 – Modeling if-conversion, implemented using a select unit. The long-latency cycle (shown in red dashed) through the select will limit achievable throughput, even if this input is never selected by the unit; hence, we omit this input from the model to place and size buffers for the best-case throughput.

It is interesting to note that this behavior corresponds to the behavior of a variable-latency unit; yet, instead of a single unit with varying latency, as we discussed in Section 3.4.3, the latency variability is now due to the different latencies of the two paths between the fork and the select (i.e., the path through the multiplier and the path through the adder). In this case, this variability cannot be resolved at unit level, i.e., adapting the timing parameters of the select unit would not impact the modeled latency of its incoming paths. Instead, we exclude from the throughput constraints the input edge of the select unit which is on the long-latency cycle, as indicated in Figure 3.13; essentially, the model will assume that this input is never taken and the best case throughput will be computed accordingly.

We model the select unit inputs as follows: (1) we include into the throughput constraints each select input edge which is on a noncyclic path, as this latency will never compromise throughput, (2) if the select has a single input on a cyclic path, we exclude the corresponding input edge from the throughput constraints, and (3) if the select has both inputs on cyclic paths, we exclude the input edge on the cycle with more sequential stages; this situation is shown in Figure 3.13.

Note that this model produces optimal throughput regardless of which input is actually taken during circuit execution, as the buffer sizes are determined based on the highest possible throughput values—the produced buffering will support lower throughputs (potentially caused by the slower input which our model ignored) as well. It is interesting to note that the same effect could be achieved by handling the choices of the select unit similar to the choices in the CFG graph, i.e., by decoupling a CFDFC with a select unit into two choice-free graphs (with each graph considering only one of the select inputs). However, this strategy would increase the

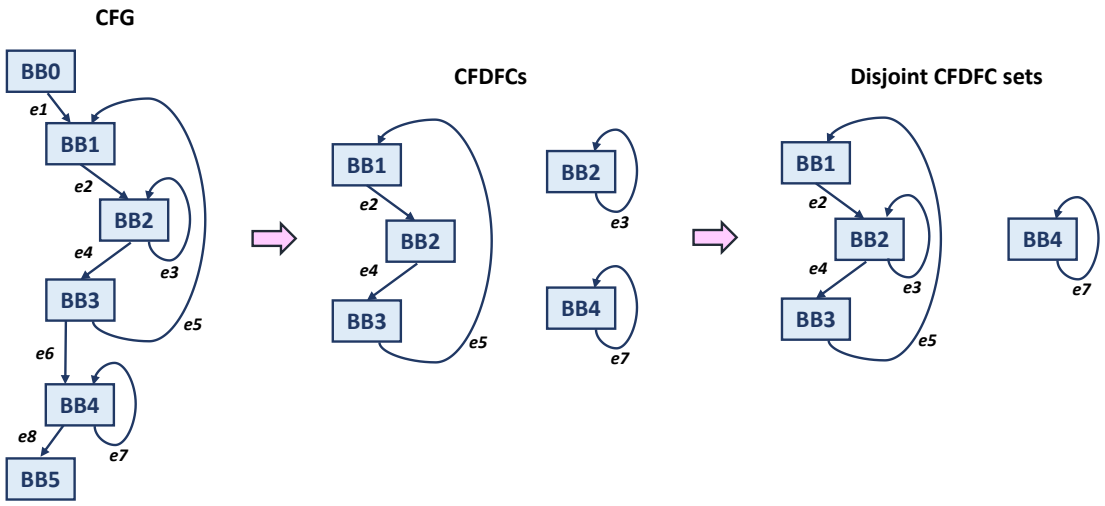


Figure 3.14 – Splitting the circuit into disjoint CFDFC sets to ensure MILP scalability. This circuit represented by the CFG in the figure consist of three CFDFCs which can be grouped into two disjoint sets. Applying the MILP on each set separately reduces the size of each MILP and decreases overall runtime.

model complexity (by adding a new set of throughput constraints per input of each select unit), while producing equivalent results.

3.5 Scalability

In this section, we discuss the runtime complexity of the MILP model described in Section 3.3.3 and propose a technique to ensure scalability when optimizing complex circuits.

The ILP for cycle extraction operates on the CFG of the program, which usually covers a limited number of BBs and control-flow edges. Hence, this ILP is typically of low complexity and size and it does not impact the overall algorithm runtime—we confirm experimentally this intuition in Section 3.6. However, the MILP for performance optimization operates on the dataflow graph of the program. While the throughput is optimized locally by applying the throughput constraints on subsets of the circuits (i.e., the frequently executed CFDFCs), the relations for path constraints (i.e., Equations 3.6 and 3.7) extend on the *entire dataflow graph*—they need to ensure that the circuit as a whole meets the target period. The MILP size and runtime is therefore dependent on the overall number of channels and units of the dataflow circuit, which can result in large runtimes when optimizing complex designs.

A possible method to limit the MILP runtime of large applications is to split the dataflow graph into disjoint sets of CFDFCs (i.e., into CFDFCs obtained from strongly connected components of the CFG which do not share any BBs or edges among each other) and to optimize them *separately* using the MILP. This procedure maximizes the throughput Θ and satisfies the period constraint $\mathbf{P}$ within the CFDFCs of each disjoint set. Afterwards, we need to ensure that the

complete circuit satisfies the period constraint. Hence, we apply the path constraints (i.e., Equations 3.6 and 3.7) on the channels and units which were not covered by any of the CFDFC sets. The buffer placement solutions (i.e., the values of R_c) from the CFDFC set optimization are now set as constants to ensure that the combinational paths across set boundaries are appropriately handled. The channels optimized in this step do not need to be subject to any throughput constraints as they are of minimal importance for the overall performance (i.e., they usually belong to paths executed only a single time as the circuit runs); the sizes of all buffers correspondingly inserted can, therefore, be set to 1 (i.e., $N_c = 1$). The cost function of this final step minimizes the number of inserted buffers:

$$\min : \sum_c R_c, \tag{3.17}$$

Figure 3.14 illustrates this approach. The dataflow circuit represented by this CFG contains three CFDFCs—two of them share BBs and need to be optimized together. The third CFDFC (corresponding to BB4 in the figure) can be optimized separately. After solving the MILP for each of the two independent CFDFC sets, their throughput Θ will be maximized and each set will meet the target period **P**. To ensure that the complete circuit respects **P**, we subsequently need to optimize the remaining parts of the dataflow circuit (in this case, the channels within BB0 and BB5, as well as those corresponding to edges *e1*, *e6*, and *e8*) using only the path constraints.

In summary, applying the MILP on disjoint CFDFC sets reduces the problem complexity while satisfying the desired clock period and achieving the same CFDFC throughput as the global MILP solution. We will show the effectiveness of this approach in Section 3.6.

3.6 Evaluation

In this section, we demonstrate the ability of our optimization technique to maximize throughput under a given clock period constraint. We compare our optimization approach with a naive buffer placement strategy, discuss the runtime of our algorithm and methods to improve it, and investigate the effectiveness of the period constraint.

3.6.1 Methodology

Our optimization strategy is summarized in Algorithm 3.1. We profile the intermediate representation of the benchmarks to obtain the execution frequencies of the CFG edges—we insert counters into the IR code to count the control-flow decisions taken in each executed BB and annotate the CFG with the obtained counts. After the IR has been transformed into a dataflow circuit, as described in Chapter 2, we use the information on execution frequencies to identify the CFDFCs using the ILP from Section 3.3.1. We then apply the MILP from Section 3.3.3 to determine the buffer placement and sizes which satisfy the target clock period and maximize the loop throughputs—we employ the cost function from Equation 3.14. The weights of each

```
// Input:  CFG (control-flow graph)
// Input:  DFG (dataflow graph)
// Output:  buffers (list of dataflow channels characterized with
// buffer capacity and transparency)

// 1.  Identify choice-free subgraphs of the dataflow graph
profile = ProfileApplication (CFG)
// ILP for iterative cycle extraction
cycles = ExtractCycles (CFG, profile)
// Find dataflow subgraph of each cycle
foreach c ∈ cycles do
    subgraph (c) = FindDataflowSubgraph (c, DFG)

// 2.  Optimize performance
foreach c ∈ cycles do
    // Choice-free subgraph throughput
    th.add(ThroughputConstraints(subgraph(c)))
foreach u ∈ DFG do
    // CP of entire dataflow graph
    cp.add(PeriodConstraints(u, e))

// MILP to max.  throughput under CP constraint
buffers = MILP (th, cp, CP_target)
```

Algorithm 3.1: Performance optimization.

throughput term are proportional to the runtime fraction of the corresponding CFDFC in the program profile and the number of units it contains (i.e., for CFDFC i, $w_i = units_i \cdot freq_i / freq_{tot}$). We choose the constant $\lambda = 10^{-5}$ to account for the minimization of buffer sizes. The MILP relies on static timing information about the unit delays—we consider exclusively the datapath of each unit. We present our results for two target periods: 4 ns and 3 ns—in the rest of this section, we denote the corresponding results as *MILP 4* and *MILP 3*.

We use ModelSim to measure throughput (represented as the average loop initiation interval, $II = 1/\Theta$) and to verify functional correctness. We target a Xilinx Kintex-7 FPGA and we use Vivado to measure the delays of the units. We obtain the clock period (CP) and the resource usage after placement and routing. We use the CBC mixed-integer programming solver [24] and measure its runtime on an Intel® Core™ i7-8550U CPU (i.e., a standard consumer laptop) at 1.80 GHz.

3.6.2 Benchmarks

We explore various kernels from literature [80, 99] and the PolyBench suite [98]; they contain pipelined computational units and exhibit different loop properties and organizations, as listed in Table 3.1: (1) *Sumi3* calculates the sum of array element cubes; it is similar to the example

Table 3.1 – Benchmark characteristics. Set and CFDFC count of our benchmarks, as well as main property of their loops.

Benchmark	Sets	CFDFCs	Property
Sumi3	1	1	regular
Fir	1	1	regular
MatVec	1	2	regular
BiCG	1	2	regular
IIR	1	1	loop-carried dependence
Cordic	1	1	loop-carried dependence
Covar	3	7	regular
Covar (f)	3	7	loop-carried dependence
Gemver	4	7	regular
CDiv	1	2	conditional execution

from Figure 3.3c. *FIR* (Finite Impulse Response), *MatVec* (Matrix-Vector Multiplication), and *BiCG* (BiCGStab Linear Solver) are regular kernels implemented as a single loop or loop nest. *IIR* (Infinite Impulse Response) and *Cordic* (COordinate Rotation DIgital Computer) have loop-carried dependences which take multiple cycles to compute and therefore limit the achievable loop initiation interval. *Covar* and *Covar (f)* implement the integer and floating-point version of the covariance computation, with and without multiple-cycle loop-carried dependences, respectively. These two benchmarks as well as *Gemver* contain multiple loop nests (i.e., multiple CFDFC sets), as indicated in the table. Finally, *CDiv* calculates a complex quotient of complex numbers—the loop contains a noninlined if-else condition (i.e., it is implemented as two CFDFCs, similar to the example in Figure 3.7); we assume a data distribution where the if-condition is taken in 55% of the total loop iterations.

3.6.3 Comparison with Naive Buffer Placement

We demonstrate the performance superiority of our optimized circuits over equivalent designs with buffers placed naively, based on an existing heuristic [73] which cuts every combinational cycle with a single buffer and does not place any FIFOs into the designs.

Tables 3.2 and 3.3 show our main results. The circuits produced using the naive strategy (i.e., *Naive*) qualitatively correspond to the circuit in Figure 3.3a: they contain the minimal number of buffers to create functional circuits (i.e., circuits with no combinational loops), but there is no way to control the critical path and backpressure significantly lowers throughput. In contrast, the designs optimized using our technique (i.e., *MILP 4* and *MILP 3*) are able to achieve maximum throughput (i.e., the best possible loop II) of the innermost loops.

The resource increase (shown in Table 3.3) is due to the additional buffers which our technique employs, as indicated under *Buffers*. The designs with high throughput require transparent buffers of larger sizes (i.e., FIFOs) to maintain the token rate; those with a lower target CP need

Table 3.2 – Timing comparison of naive and optimized dataflow circuits. Dataflow circuits optimized with our strategy, *MILP 4* and *MILP 3*, with a target CP of 4 ns and 3 ns, respectively, compared to a naive buffer placement (*Naive*). The II column indicates the initiation intervals of the innermost loops. The rightmost column indicates the MILP runtime (the value of 3600 indicates a timeout of 1 hour).

Benchmark	Method	CP (ns)	II= 1/Θ	Execution Time (μs)	Speedup	MILP Runtime (s)
Sumi3	Naive	4.3	10	43.0	–	–
	MILP 4	4.0	1	4.1	**10.6**×	0.1
	MILP 3	3.5	2	7.1	**6.1**×	1.2
FIR	Naive	4.3	6	25.8	–	–
	MILP 4	3.6	1	3.6	**7.2**×	0.1
	MILP 3	3.5	2	7.0	**3.7**×	0.8
MatVec	Naive	5.9	6	31.9	–	–
	MILP 4	4.0	1	3.6	**8.9**×	15.7
	MILP 3	3.9	2	7.3	**4.4**×	25.0
BiCG	Naive	6.0	6	32.4	–	–
	MILP 4	5.9	1	6.4	**5.0**×	1328.4
	MILP 3	4.1	2	7.7	**4.2**×	2195.5
IIR	Naive	5.9	6	35.4	–	–
	MILP 4	3.9	5	19.5	**1.8**×	1.1
	MILP 3	3.4	5	17.0	**2.1**×	20.1
Cordic	Naive	5.8	20	116.1	–	–
	MILP 4	4.5	20	87.8	**1.3**×	8.1
	MILP 3	5.0	20	97.6	**1.2**×	8.1
Covar	Naive	6.8	2, 4, 4	698.5	–	–
	MILP 4	6.5	1, 1, 1	197.5	**3.5**×	3600
	MILP 3	5.6	2, 2, 2	182.9	**3.8**×	3600
Covar (f)	Naive	7.1	11, 11, 17	1833.6	–	–
	MILP 4	7.2	11, 1, 11	1057.2	**1.7**×	3600
	MILP 3	5.4	11, 2, 11	865.7	**2.1**×	3600
Gemver	Naive	7.7	6, 10, 2, 10	180.7	–	–
	MILP 4	7.4	1, 1, 1, 1	23.5	**7.7**×	3600
	MILP 3	5.5	2, 2, 2, 2	31.3	**5.8**×	3600
CDiv	Naive	10.5	40, 40	787.9	–	–
	MILP 4	7.5	3, 3	23.3	**33.8**×	25.4
	MILP 3	7.2	5, 5	36.8	**21.4**×	153.9

more nontransparent buffers to cut the combinational paths but use smaller buffer sizes due to the lowered throughput (consider, for instance, the buffer sizes in the *MILP 4* and *MILP 3* solutions of *Sumi3*). Setting a low target CP degrades throughput (as it requires the insertion of multiple nontransparent buffers on cyclic paths) and, consequently, performance, in all applications but the *IIR*, *Covar* and *Covar (f)*. In these applications, the throughput is dictated by the pipelined units on the cyclic paths and not influenced by the additional buffers, so the total execution time benefits from the lowered CP. The discrepancies between the target and achieved CP are largely due to the timing variations caused by FPGA place-and-route. Our timing model could be further refined for greater accuracy without any qualitative change (e.g., by including setup delays of the buffers and considering the impact of control paths).

Table 3.3 – Resource comparison of naive and optimized dataflow circuits. LUTs, FFs, and DSPs of circuits optimized with our strategy, *MILP 4* and *MILP 3*, with a target CP of 4 ns and 3 ns, respectively, compared to a naive buffer placement (*Naive*). The types of instantiated buffers are shown under *Buffers* (e.g., 3 N2-4 indicates the usage of 3 nontransparent buffers with two to four slots).

Benchmark	Method	LUTs	FFs	DSPs	Buffers
Sumi3	Naive	287	331	6	3 N2
	MILP 4	402 (+40%)	403 (+22%)	6	8 N1-2, T4, 2 T9
	MILP 3	413 (+44%)	522 (+58%)	6	10 N1-2, 2 T1-2, 2 T5
FIR	Naive	380	384	3	3 N2
	MILP 4	463 (+22%)	526 (+37%)	3	5 N1-2, 2 T6-7
	MILP 3	628 (+65%)	688 (+79%)	3	7 N1-2, 2 T4-5
MatVec	Naive	626	517	3	6 N2
	MILP 4	843 (+35%)	631 (+22%)	3	11 N1-2, 5 T3-8
	MILP 3	947 (+51%)	849 (+64%)	3	16 N1-2, N4, 6 T1-3
BiCG	Naive	831	758	6	6 N2
	MILP 4	1144 (+38%)	1157 (+53%)	6	16 N1-3, 7 T1-3, 4 T5-7
	MILP 3	1140 (+37%)	1255 (+66%)	6	14 N1-2, 2 N4-5, 8 T1-3
IIR	Naive	648	663	6	5 N2
	MILP 4	745 (+15%)	1096 (+65%)	6	10 N1-2, 6 T1-2
	MILP 3	772 (+19%)	1094 (+65%)	6	12 N1-2, 5 T1-2
Cordic	Naive	1950	2754	24	7 N2
	MILP 4	2075 (+6%)	3086 (+12%)	24	16 N1-2, 9 T1-2
	MILP 3	2145 (+10%)	3016 (+10%)	24	17 N1-2, 9 T1-2
Covar	Naive	2347	1801	3	23 N2
	MILP 4	3882 (+65%)	3024 (+68%)	3	44 N1-3, 16 T1-3, 6 T4-19
	MILP 3	3953 (+68%)	3388 (+88%)	3	54 N1-3, 20 T1-3, 3 N4-9
Covar (f)	Naive	3493	3795	9	23 N2
	MILP 4	4298 (+23%)	4727 (+25%)	9	43 N1-3, 18 T1-3, 3 N6-10, 4 T6-20
	MILP 3	4558 (+30%)	5196 (+37%)	9	46 N1-3, 24 T1-3, 2 N5-6, 6 T4-13
Gemver	Naive	3098	2903	18	30 N2
	MILP 4	4076 (+32%)	3990 (+37%)	18	60 N1-3, 7 T1-3, 8 N5-12, 5 T4-10
	MILP 3	4066 (+31%)	4353 (+50%)	18	58 N1-3, 29 T1-3, 3 T4-7, 2 N5-7
CDiv	Naive	14461	14081	18	6 N2
	MILP 4	15197 (+5%)	14780 (+5%)	18	11 N1, 6 T1-2, 8 T11-26, 8 N13-26
	MILP 3	15164 (+5%)	14946 (+6%)	18	12 N1, 8 T1, 16 T6-16

3.6.4 MILP Runtime Analysis

The rightmost column of Table 3.2 reports the runtime of the MILP for performance optimization. In all our benchmarks, the runtime of the ILP for extracting the CFDFC was negligible (i.e., less than 0.1 *s*) in comparison to the MILP runtime. It is evident from the table that the MILP runtime significantly depends on the size and complexity of the application—larger applications need a prolonged MILP runtime because the MILP covers the units and channels of the entire dataflow graph, as discussed in Section 3.5.

MILP solvers tend to find an acceptable solution (i.e., a near-optimal cost function value) early on and then spend a long time attempting to improve it. This effect is evident from Figure 3.15a, which shows the obtained cost function value (i.e., the sum of the weighted CFDFC throughputs, as given in Equation 3.14), relative to the optimal cost function value for a given target CP. The graph depicts only the results of the benchmarks which take longer than 1 second to converge

to the optimum value of 1. While it is clear that the convergence time is lower than the overall MILP runtime reported in Table 3.2, it is still nonnegligible in certain cases (e.g., *Gemver* requires at least 30 minutes of MILP runtime to find a good solution).

We investigate the effectiveness of the heuristic from Section 3.5 to reduce the MILP runtime. We organize the CFDFCs into independent sets and employ the MILP on each set separately. The results we obtain are plotted in Figure 3.15b which compares the obtained cost function result to the optimal result, exactly as in the previous graph. Contrasting the two graphs indicates that this method successfully lowers the time needed for the MILP to converge.

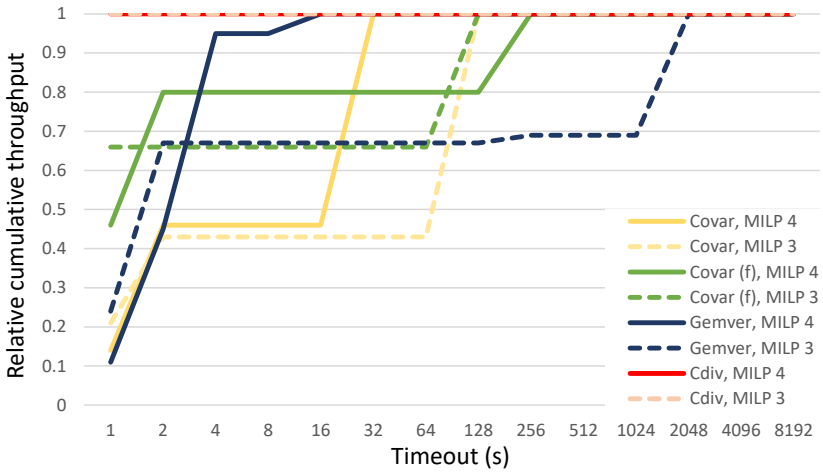
The two versions of the MILP which we have considered so far optimized *all* CFDFCs of the program. Our next experiment is based on the intuition that some CFDFCs do not contribute significantly to the execution time of the application (e.g., the outermost loop of a nested loop)—they can be removed from the cost function without a notable performance penalty. We demonstrate this effect in Figure 3.15c, where we compare the cost value of the MILP which optimizes the throughput of a *single*, most important CFDFC *per set*, with the optimal MILP cost value, as in the previous graphs. This MILP converges rapidly and, in most cases, obtains a near-optimal value, as the removed CFDFCs contributed to the cost function with a negligible weight factor. However, some applications such as *CDiv* suffer from this simplification: this application has two CFDFCs with similar contributions (i.e., 55% and 45%) to execution time; optimizing the throughput of only one CFDFC lowers the obtainable cost function value and, consequently, application performance.

The results of our runtime analysis can, therefore, be summarized as follows: (1) it seems possible to rely on timeouts to find good solutions in reasonable runtime, (2) the heuristic from Section 3.5 helps in further reducing the MILP runtime, and (3) not all CFDFCs play an important role in achievable application performance; it is possible to simplify the MILP to account for this fact and to further reduce its runtime at a negligible penalty.

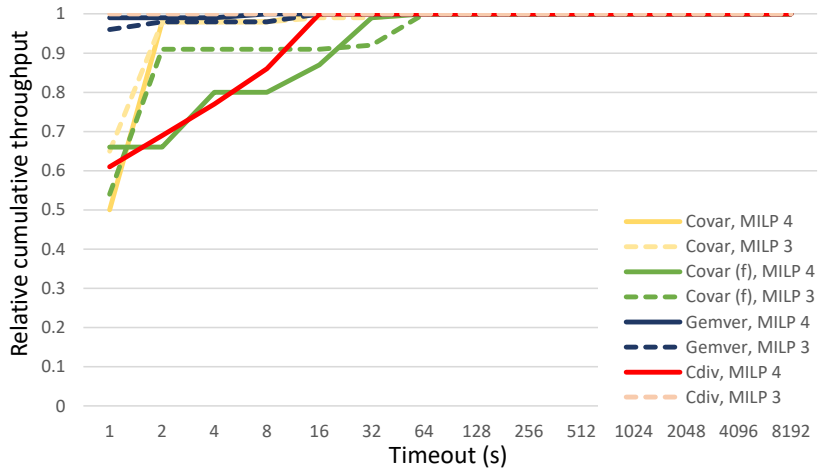
3.6.5 Comparison of MILP Solutions

To complement our runtime analysis from the previous section, we evaluate the quality of solutions obtained in the following manner: (1) we choose a timeout of 1 minute to terminate the MILP, (2) we split the CFDFCs into sets to employ the heuristic from Section 3.5, and (3) in the cost function of each set, we include all CFDFCs, starting from the one with the highest weight, until there is at least an order of magnitude difference in the cost term weight between the last one included and the first one not included. This ensures that the most relevant CFDFCs of each set are optimized (e.g., the innermost loops of our benchmarks; in *CDiv*, this approach includes the throughput optimization of both the if and the else branch).

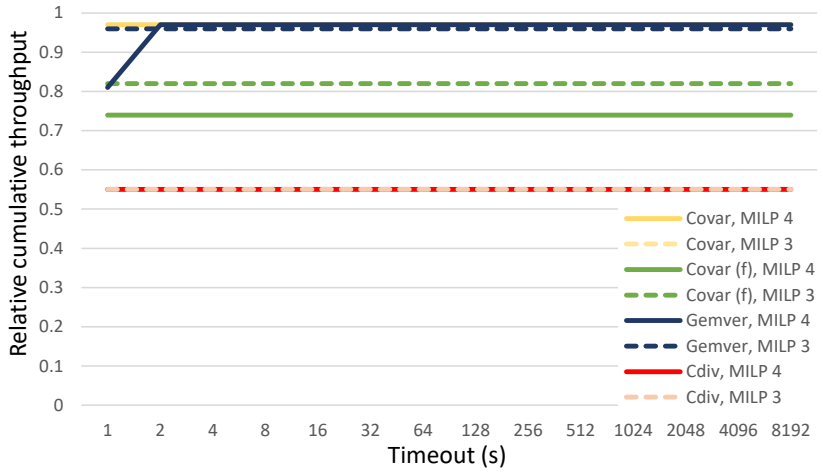
Figure 3.16 shows the cycle count, total execution time, and resource consumption of the solutions obtained in such a manner, relative to the optimal MILP solutions from Tables 3.2 and 3.3. In applications which have a single CFDFC per set, the obtained cycle count is equal



(a) Full MILP (i.e., single MILP for entire application).

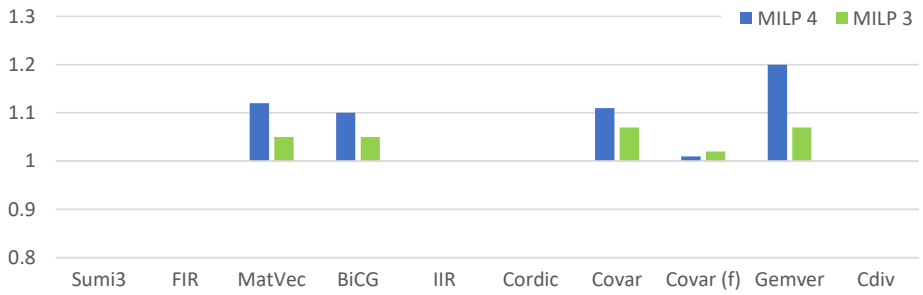


(b) MILP in CFDFC sets.

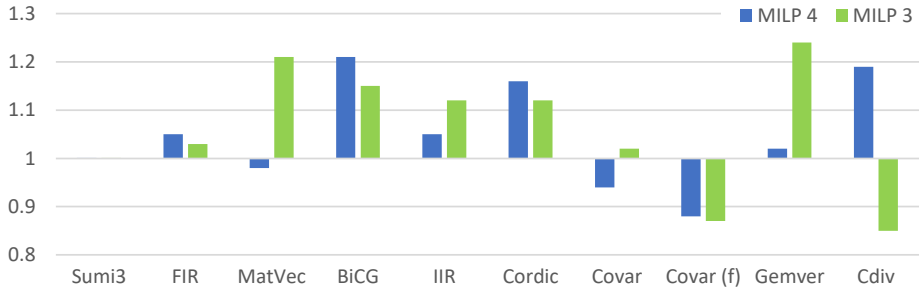


(c) MILP in CFDFC sets, single CFDFC per set.

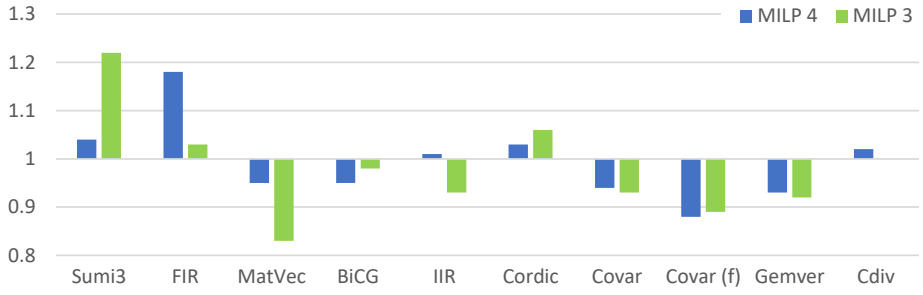
Figure 3.15 – Runtime comparison of the full MILP with the MILP applied on individual CFDFC sets.



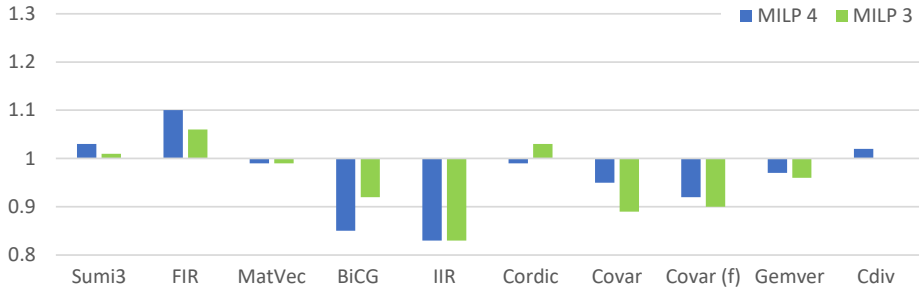
(a) Cycle count ratio (MILP in sets/full MILP).



(b) Execution time ratio (MILP in sets/full MILP).



(c) LUT count ratio (MILP in sets/full MILP).



(d) FF count ratio (MILP in sets/full MILP).

Figure 3.16 – Comparison of solutions obtained by applying the MILP on individual CFDFC sets with the optimal MILP solutions. The optimal solutions are obtained by employing the MILP on the entire circuits, as shown in Tables 3.2 and 3.3.

Table 3.4 – Timing and resources of kernels which contain computational units with variable latency and II, as well as if-conversion. We compare kernels optimized with our strategy (*MILP 4*) to those with a naive buffer placement (*Naive*). The II and execution time are shown as a range from the best-case to the worst-case behavior, as determined by data dependences.

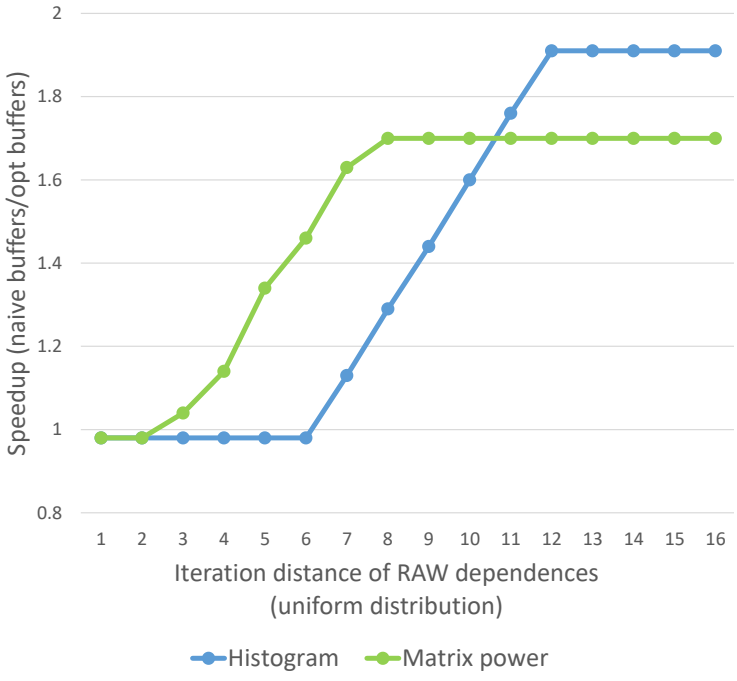
Bench-mark	Method	CP (ns)	II= 1/Θ	Execution Time (μs)	Speedup	LUTs	FFs	DSPs
Histogram	Naive	6.1	2.0-12.0	12.2-73.2	–	16627	3529	2
	MILP 4	6.3	1.0-12.0	6.4-75.6	**1.9 – 1.0**×	16879	3562	2
Matrix power	Naive	6.0	3.5-11.7	8.1-26.7	–	16870	3696	5
	MILP 4	6.2	2.0-11.5	4.9-27.2	**1.7 – 1.0**×	16955	3744	5
If loop add	Naive	4.9	12.0-20.0	58.8-98.0	–	903	1284	4
	MILP 4	5.0	1.0-10.0	5.1-50.1	**11.6 – 2.0**×	960	1318	4
If loop mul	Naive	5.0	12.0-16.0	60.0-80.0	–	858	1091	5
	MILP 4	5.2	1.0-6.0	5.3-31.3	**11.4 – 2.6**×	892	1127	5

to the optimal because our heuristic covers the entire application; in others (i.e., applications with nested loops) the count slightly increases because the throughput of the outer loops is not optimized. The total execution time varies due to the changes in obtained frequency (largely caused by FPGA place-and-route, as discussed earlier). In most cases, these solutions require fewer resources than the optimal MILP solutions—as the throughputs of certain loops are not optimized, fewer FIFOs are instantiated. All these variabilities are expected and in an acceptable range for the significant MILP runtime reduction which this heuristic approach offers.

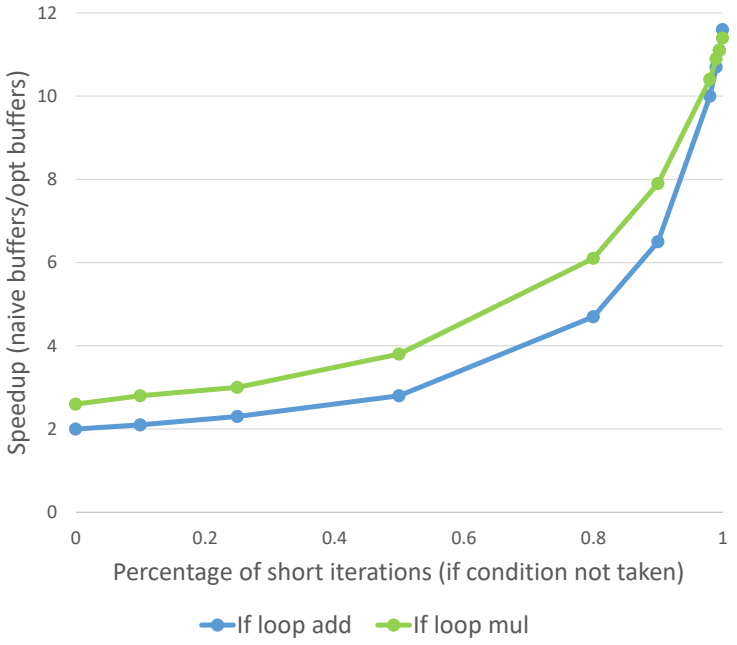
3.6.6 Variable Latency, II, and If-Conversion

In this section, we investigate the effectiveness of our method to model units with variable latency and II, as well as if-conversion. The benchmarks we evaluate exhibit data-dependent and variable behavior: *Histogram* and *Matrix power* have memory access patterns which cannot be determined at compile time; as we will discuss in Chapter 5, they require a load-store queue at the memory interface, which exhibits variable latency and II depending on the runtime-determined data dependences. *If loop add* and *If loop mul* have a potential dependence across loop iterations; the data to send to the following iteration is selected using a select unit based on a data-dependent condition, determined during program execution; *If loop add* is the kernel from Figure 2.3 and *If loop mul* differs only in the conditional computation. All these kernels represent situations where dataflow circuits excel in contrast to statically scheduled HLS circuits.

We follow the rules from Sections 3.4.2, 3.4.3, and 3.4.4 to model the behavior of these kernels and compare them to the same kernels where buffers are placed naively. Our results are shown in Table 3.4. As the average II and execution time in all kernels depend on the input data, we indicate these values as a range from their minimum to their maximum value. In the kernels with the LSQ, the best-case scenario (i.e., with the smallest II and execution time) corresponds to a situation where there are no RAW dependences among loop iterations; in the kernels with



(a)



(b)

Figure 3.17 – Speedup of the optimized kernels with respect to the naive kernels for varying data and control dependences. In Figure 3.17a, we change the number of iterations between dependent read and write accesses. In Figure 3.17b, we change the percentage of short loop iterations (i.e., the percentage of loop iterations where the long-latency if-condition is not taken).

the select unit, the same is achieved when there are no loop-carried dependences. The other extreme is obtained when every pair of loop iterations has a RAW dependence and a loop-carried dependence, respectively. All other possible data points fall in the middle of the shown ranges.

In Figure 3.17, we show the speedup (i.e., execution time ratio) of the optimized kernels with respect to the naive kernels for a varying number of data or control dependences (i.e., the data points which are within the execution time range shown in Table 3.4). In Figure 3.17a, we explore the execution time of *Histogram* and *Matrix power* for a varying number of iterations between dependent read and write accesses (e.g., distance 2 indicates that a load reads a value which is stored into memory two iterations ago). In Figure 3.17b, we explore *If loop add* and *If loop mul* while changing the percentage of short loop iterations, i.e., the iterations where the if-condition is not taken and there is no loop-carried dependence.

As Table 3.4 and Figure 3.17 illustrate, the naive technique achieves only limited pipelining, whereas the kernels optimized with our approach are able to achieve the highest possible throughput. In these examples, the achievable throughput is variable and dependent on the actual data dependences: When a large number of dependences is present, the II increases to honor them; still, the optimized kernels typically achieve higher throughput than the naive kernels, which suffer from backpressure and suboptimal buffering even in those cases. As the number of dependences decreases (i.e., the iteration distance between RAWs or the percentage of short iterations grows), the kernels become more pipelineable; the optimized designs benefit from the appropriate buffering to achieve better performance and larger speedups. When there are no dependences, all optimized kernels achieve the best possible II.

The resource trends in Table 3.4 follow the ones we discussed in Section 3.6.3 and depend on the number and the capacity of the inserted buffers; the overheads of our technique are minor and probably acceptable for the significant performance benefits, which indicates that our technique is critical to truly capture the variable and dynamic behavior of dataflow circuits.

3.6.7 Effectiveness of the CP Constraint

In this section, we further explore the capabilities of our model to control the critical path. We analyze the effects of the CP constraint on an unrolled accumulator, implemented as a binary tree of adders with 16 inputs; this example gives more room for CP exploration than the benchmarks from the previous section. We present the results in Table 3.5. The naively obtained CP corresponds to the combinational path through the entire adder tree. Lowering the constraint inserts buffers between different tree stages. Although the achieved CP tracks well the constraint in most cases, the maximum frequency cannot be reached. This effect is most likely due to the control paths which are not included in our timing model and become dominant with tighter CP constraints. Our timing model could be further refined to account for these effects as well.

Table 3.5 – Exploration of the effectiveness of the clock period (CP) constraint on a tree of combinational adders.

Target CP (ns)	Achieved CP (ns)	II = 1/Θ	Execution Time (μs)	LUTs	FFs
–	9.1	2	15.7	1578	1665
8	7.7	1	7.7	1632	1742
6	5.7	1	5.7	1661	1810
4	4.1	1	4.1	1853	2084
3	3.6	2	7.2	2188	2696

3.7 Conclusions

In this chapter, we presented a performance optimization model for dataflow circuits obtained out of C code. Our mixed-integer linear programming model is based on the theory of marked graphs and allows for resource-optimal buffer placement and sizing, with the purpose of maximizing throughput at the desired clock frequency. In addition to the exact model formulation, we described a computationally-efficient heuristic which achieves near-optimal results; its ability to handle large benchmarks makes our approach applicable to real-world and complex workloads. On benchmarks obtained out of C code, we demonstrated the ability of our approach to achieve high-throughput, pipelined dataflow circuits. We showed that our approach effectively handles different HLS features such as pipelined computational units, variable-latency memory interfaces, and if-converted code. This optimization is one of the key steps in making dynamic scheduling truly competitive with existing HLS techniques.

4 Resource Sharing in Dataflow Circuits

Now that our circuits are able to achieve high-throughput pipelines, we address another important optimization aspect: sharing functional units to save resources. As discussed in Section 2.4.2, the scheduling flexibility of dataflow circuits makes sharing challenging: in the absence of a predetermined schedule, the cycle in which each operation executes is unknown. Hence, dataflow approaches typically employ an individual unit for each operation and result in area-expensive solutions. The intuition on how to implement sharing in a dataflow context is fairly straightforward: instead of relying on cycle information on operation execution, one could consider statistical information on unit utilization—if a certain unit is, on average, underutilized (i.e., not always busy computing), it may be possible to share it with another underutilized unit. However, on its own, this strategy does not consider two important concerns: (1) Sharing may compromise some of the fundamental functional properties of dataflow circuits; one needs to ensure that the resulting circuits are always deadlock-free. (2) Sharing may postpone the execution of some operation with respect to its execution in the original dataflow circuit and, consequently, compromise performance; one needs to evaluate and minimize this performance impact.

In this chapter, we present a complete methodology to implement resource sharing in dataflow designs. We illustrate the difficulties of performing resource sharing in the context of dataflow circuits; we formulate the necessary requirements to ensure deadlock-free execution and implement a sharing mechanism accordingly. We then discuss how to appropriately model and minimize the impact of delays caused by sharing. Finally, we show that our technique results in up to 72% DSP reduction with minimal or no impact on execution time compared to dataflow circuits which do not implement resource sharing.

4.1 Motivation

To illustrate the challenges of resource sharing in dataflow circuits, we revisit the code from Figure 2.11 in Figure 4.1. The execution of this circuit starts when a token enters through the starting point; a new loop iteration is triggered as soon as a token reenters the loop body through

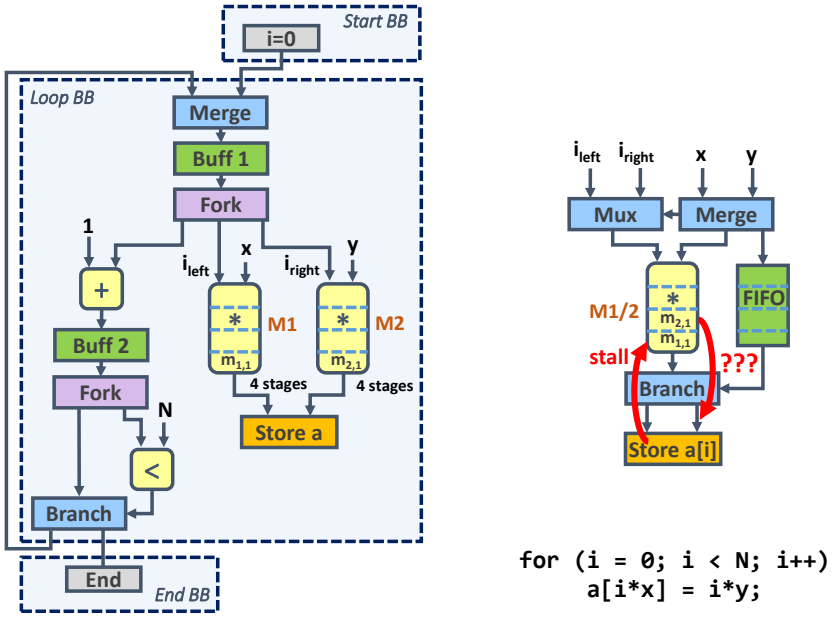


Figure 4.1 – Dataflow circuit and a possible implementation of resource sharing. The two multiplications of the loop could be computed using a single multiplier, with multiplexing logic at its inputs and its output. However, this mechanism on its own does not guarantee that the circuit is deadlock-free nor that its performance is optimal. The multiplication results are indicated as $m_{op,iter}$—e.g., $m_{2,1}$ is the result of operation M2 from loop iteration 1.

the cyclic path (in this example, every second clock cycle because of two buffers, Buff 1 and Buff 2, on the cyclic path through the merge and the branch).

The loop in the figure contains two pipelined, 4-stage multipliers; all other units, apart from the buffers, are combinational. Since a new loop iteration starts every second cycle, the two multiplications of the loop could be performed using a single multiplier. A simple and intuitive implementation is given on the right of the figure—the merge and mux units are used to steer one set of input tokens at a time into the shared unit (as indicated in the figure, these units must communicate to ensure that they always accept the matching operands from their predecessors). The branch at the output ensures that the result is sent to the appropriate successor, depending on the origin of the operand tokens—this information is conveyed to the branch by the input merge through a FIFO.

Surprisingly, this implementation does not guarantee a functional circuit: in this example, the store needs both operands (i.e., both the address and the data) to execute; it therefore stalls the available operand ($m_{1,1}$, i.e., the result of M1 of the first iteration) while it waits for the second operand ($m_{2,1}$, i.e., the result of M2). However, because of the stall of $m_{1,1}$, $m_{2,1}$ will never be able to exit the shared unit and arrive to the store, therefore causing deadlock.

Such problems are absent by construction in elementary dataflow circuits, described in Chapter 2, where each operation uses an individual unit and only a single token per loop iteration is transferred from one unit to another. However, introducing sharing compromises this property;

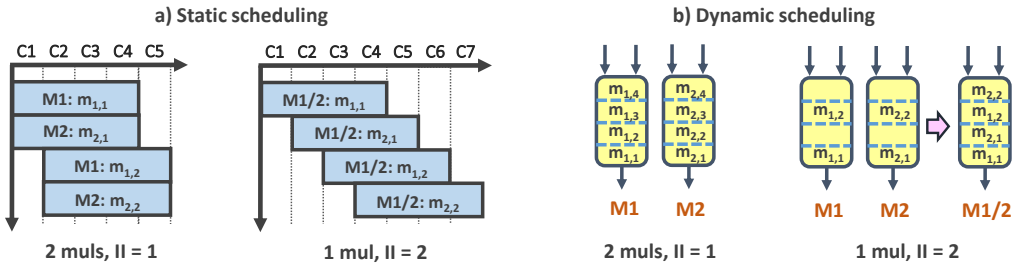


Figure 4.2 – Resource sharing in static and dynamic scheduling. In static scheduling, resource sharing is decided based on the cycle information on operation execution; in contrast, dataflow circuits can rely on average unit utilization to identify good sharing candidates.

it is crucial that we develop a sharing mechanism that handles this issue and ensures the absence of deadlock in every possible case.

4.2 Deciding What to Share in a Dataflow Circuit

Standard, statically scheduled HLS tools [119, 18] perform scheduling in conjunction with resource allocation and sharing [120]; depending on the optimization objective, they trade-off area and performance by deciding the cycle in which each operation executes and allocating units accordingly. Figure 4.2a shows two possible schedules for the code from Figure 4.1: the first achieves the ideal loop pipeline with an II of 1 by scheduling both multiplications in the same cycle, hence employing two multipliers; the second increases the II to 2 and schedules each multiplication on every second cycle, which allows the usage of a single multiplier.

Dataflow circuits face the same optimization objectives and area-performance trade-offs; however, there is no predetermined schedule and no information on when each operation executes to decide how many units to employ. Instead, one can identify units which are, on average, underutilized, and use this information to implement sharing.

Several authors have discussed techniques for analyzing the timing of dataflow circuits [77, 13, 102, 101, 16]; some determine the rate at which dataflow units compute and can directly provide the information on average unit utilization. We here rely on our performance optimization approach presented in the previous chapter, which maximizes the throughput of each CFG cycle by appropriately placing and sizing buffers. The approach calculates the average *occupancy* of each unit with tokens, i.e., for a given throughput of a CFG cycle, determines the average number of tokens that each unit holds in the steady state of the cycle execution (see Section 3.3.3). We can use this information to identify good candidates for sharing: if the sum of the tokens in two units of the same type is at most equal to the unit latency (i.e., number of sequential stages), it may be possible for the operations to use a single unit without damaging the throughput of the CFG cycle. Figure 4.2b illustrates this reasoning: if the two multipliers are fully utilized, sharing cannot be implemented without a throughput penalty; however, if they are only half-utilized, one could employ a single multiplier instead.

The dataflow circuit in Figure 4.1 exhibits the behavior on the right of Figure 4.2b: Because the cyclic path contains two buffers, a new token enters the loop on every second cycle, i.e., the achieved throughput is 1/2. It is then evident that a new token enters each multiplier on every second cycle as well; in the steady state of the system, each multiplier holds two tokens and has two empty slots (i.e., the occupancy of each multiplier is equal to 2). It is therefore possible to implement the two multiplications using a single multiplier which will accept a new token and start a new multiplication on every cycle—this multiplier will, in the steady state, contain a token in every pipeline stage (i.e., its occupancy will be equal to 4) and the shared unit will always be busy computing.

Such a performance analysis approach can ensure that each shared unit receives tokens at a rate at which it is able to compute. However, this analysis on its own does not recognize that sharing may postpone a specific computation. For instance, in Figure 4.1, prior to sharing, both multiplications execute simultaneously (i.e., both $m_{1,1}$ and $m_{2,1}$ are computed at the same time by the two multipliers); with sharing, one multiplication is delayed by one clock cycle (in the right of Figure 4.2b, the multiplication $m_{2,1}$ is computed one cycle after $m_{1,1}$). In certain cases, such delays may compromise throughput, as we will discuss later. More importantly, as indicated in Section 4.1, nothing in this analysis guarantees that the dataflow circuit with sharing is deadlock-free—addressing this issue is the main contribution of this chapter.

4.3 Resource Sharing in Dataflow Circuits

In this section, we present our methodology to implement resource sharing in dataflow circuits. In Section 4.3.1, we show how to ensure that the circuit is functional when sharing operations in a straight datapath (i.e., a single BB execution). However, this mechanism does not guarantee a functional circuit when operations repeat (i.e., loops); thus, we generalize our approach in Section 4.3.2. We further adapt our solution to address performance concerns in Section 4.3.3 and discuss possible extensions of our scheme in Section 4.3.4.

4.3.1 Sharing in Straight Datapaths

Sharing requires steering data into a unit from multiple predecessor units as well as sending the output to the appropriate successor unit. This behavior can be realized as on the left of Figure 4.3, which repeats the situation from Figure 4.1: the input of the shared unit has a merge for one its operands and a mux for every other operand; the merge and muxes have as many data inputs as there are shared operations. The merge indicates to the muxes and the output branch which operand it took so that they can choose the corresponding operands and dispatch the result to the correct successor, respectively. The merge and the branch communicate through a FIFO, which has as many slots as there are pipeline stages in the unit.

Yet, as we have discussed before, this scheme is not sufficient to guarantee a functional circuit: one token may be stalled inside the unit and prevent the others from exiting, potentially causing

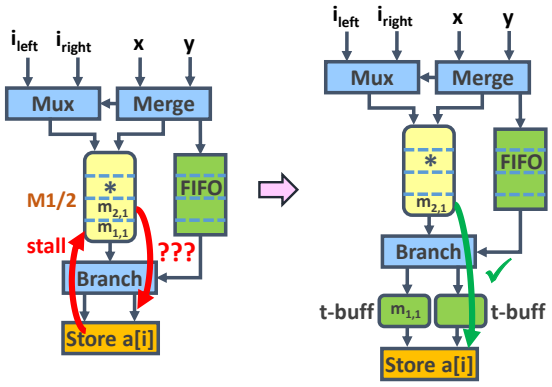


Figure 4.3 – Hardware for sharing. A naive implementation (left) results in deadlock. Placing transparent buffers (t-buffs) on the branch outputs (right) allows all tokens from a single datapath execution to exit the shared unit and all the computation of the datapath to complete.

deadlock. In this example, as the successor unit needs to join tokens $m_{1,1}$ and $m_{2,1}$ to start computing, $m_{1,1}$ cannot exit the unit until $m_{2,1}$ arrives; however, the exact same token ($m_{1,1}$) is blocking $m_{2,1}$ from ever exiting the unit and reaching the store unit, therefore infinitely starving the store and blocking the shared multiplier from processing other tokens.

The mechanism on the right of Figure 4.3 guarantees that all tokens from a straight datapath (i.e., a single BB or a sequence of nonrepeating BBs) which enter the shared unit are able to exit it by adding a 1-slot transparent buffer (i.e., *t-buff*) at each branch output. In a single BB execution, each dataflow edge transfers a single token; the output edge of the shared unit, on the other hand, does not honor this property (i.e., it transfers as many tokens as there are shared operations), but it sends only one token to each branch output (corresponding to each individual operation in the original circuit). Hence, the *t-buff* is sufficient to ensure that each token can always exit the unit, regardless of the availability of the successor (i.e., if the successor is not ready, the token will be stored in the *t-buff*; otherwise, it will be immediately sent further, as described in Section 3.1.1). No token will therefore be stalled in the unit, nor will it block other tokens in the unit; all successor units of the same BB will be able to receive their data and all BB computation will successfully complete, exactly as it would if no units were shared.

However, as we will see shortly, this hardware does not guarantee deadlock-free execution in cases where BBs repeat, as in a loop.

4.3.2 Sharing in General Datapaths

The methodology from the previous section guarantees that the circuit is functional only when sharing within a single BB or a loop iteration; we here extend this implementation to general programs.

Figure 4.4 shows two examples where the mechanism from Section 4.3.1 still does not manage to prevent deadlock: (1) Circuit 1 has a similar problem as discussed before, but occurring across

loop iterations: a token from a successive iteration ($m_{1,2}$) blocks the token from the previous iteration ($m_{2,1}$) from exiting the shared unit; at the same time, $m_{1,2}$ cannot proceed before the previous computation completes, so both tokens remain stalled in the shared unit indefinitely. (2) Circuit 2 has a cyclic path from the output of the shared unit to its input. The unit may fill with tokens and cause deadlock because there is no empty space for the tokens to move (i.e., the property which guarantees the absence of deadlock, outlined in Section 3.1.3, is violated, as no buffer slot on the cycle is empty): the token in the unit ($m_{1,2}$) needs to move into *t-buff* on the cycle, but the token in the *t-buff* ($m_{1,1}$, corresponding to input z of M2) cannot move back into the unit before another token exits.

Both problems are due to tokens entering the shared unit in an order different than the one specified by the control flow of the program—some tokens enter the unit before all tokens from the predecessor BBs have been sent into the unit and prevent the computation from the preceding BBs to complete: (1) In circuit 1, instead of consecutively consuming both tokens from the same BB execution, the unit inputs some tokens from the following loop iteration (i.e., the next BB execution) which then prevent one of the previous tokens from ever exiting the unit. (2) In circuit 2, the token that belongs to the first BB execution (i.e., first loop iteration) comes from the shared unit itself. However, instead of consuming this token to execute the first multiplication of M2, the shared unit keeps taking tokens from the following iterations (coming from the noncyclic path and performing several multiplications of M1), therefore filling the unit and preventing the older token from the *t-buff* from propagating further.

The solution to both problems is to send tokens to the shared unit in the order specified by the control flow (i.e., program order): once the execution of a BB is decided, all tokens from this BB must be consumed by the shared unit before the tokens from the following BB are allowed to enter. If all tokens from a BB are injected into the unit before any successive tokens, they are guaranteed to exit the unit, as described in Section 4.3.1. Always sending tokens into the unit in order of BB execution ensures that this property holds for any BB and any number of BB executions, therefore guaranteeing the absence of deadlock.

To implement this behavior, we extend the circuit with a specialized mechanism which enforces the ordering of tokens from different BBs as they enter the unit, as illustrated on the right of Figure 4.4; we will detail its implementation in the following sections.

4.3.3 Sharing and Performance

The previous section highlighted the need to order tokens from different BBs as they enter a shared unit to prevent deadlock. The ordering of tokens from the same BB does not compromise the functioning of the circuit, but it may impact performance.

In a circuit with sharing, one needs to make sure that the latency of a throughput-critical cycle is not increased by stalls introduced by the shared unit. Figure 4.5 illustrates this issue. In the shown dataflow circuit, both multiplications execute simultaneously; M1 is on a loop which

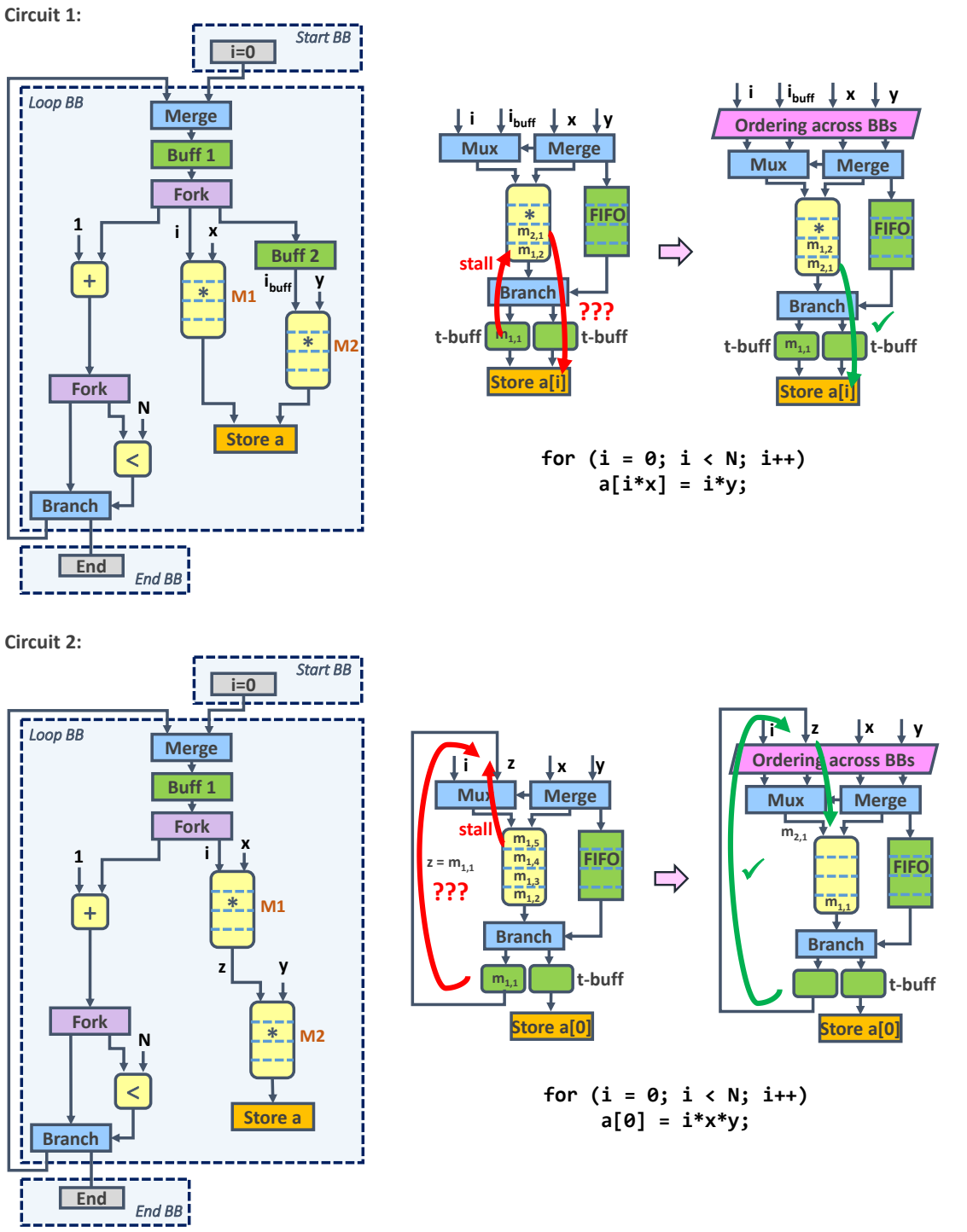


Figure 4.4 – Deadlock situations. In the shown dataflow circuits, deadlock can occur due to the reordering of tokens from different BB executions. In circuit 1, the token from the second loop iteration ($m_{1,2}$) precedes the token from the first iteration ($m_{2,1}$) and prevents it from ever exiting the unit. In circuit 2, tokens from multiple iterations may enter the unit before the second multiplication (M2) of the first iteration issues; the unit fills with tokens and no token can move forward. The solution in both cases is to force tokens to enter the unit in the order of BB execution, i.e., all tokens from one BB must enter the unit before the tokens from the successor BBs, as shown in the rightmost figures.

determines the throughput, equal to 1/5 (because of the buffer and the 4-stage multiplier on the cycle). If the two multiplications share a single unit, one of them will be postponed for a clock cycle while the multiplier consumes the inputs of the other. If the delayed computation is M1, the latency of the loop increases and, consequently, lowers the throughput to 1/6.

Therefore, in addition to enforcing ordering of operations from different BBs, as previously described, one could order operations within each BB as well, as suggested on the right of Figure 4.5, such that the throughput impact is minimal. We incorporate this notion into our sharing strategy, as we will describe in Section 4.5: for every possible ordering of shared operations, we use the performance analysis from Chapter 3 to determine the achievable throughput as well as to obtain the optimal buffer placement and sizes for the particular configuration—this exploration allows us to choose an ordering with the least impact on performance.

Note that we now implement a total order of the operations and, therefore, the corresponding operands always arrive aligned to the unit; hence, the muxes at the unit inputs (see Section 4.3.1) can be replaced by merges. This will be particularly clear in our implementation of the ordering logic in Section 4.4.1 and Figure 4.6.

4.3.4 Extending the Ordering Scheme

The ordering rules that we have described so far ensure the absence of deadlock by ordering tokens across BBs (Section 4.3.2); to ensure the best possible throughput in the presence of such ordering, we order operations within a BB as well (Section 4.3.3). Interestingly, ordering tokens across BBs may, in particular cases, lower the throughput of a loop, as it may limit the overlapping of operations from different loop iterations. This is the case in the second circuit from Figure 4.4: One could, in principle, implement sharing for M1 and M2 with a throughput of 1/2 (i.e., an II of 2) by starting one of the two multiplications on every consecutive clock cycle. However, our strategy from Section 4.3.2 lowers the throughput to 1/5—as suggested on the right of Figure 4.4, the first computation of M2 ($m_{2,1}$) starts 4 cycles after the start of the first computation from M1 ($m_{1,1}$); the next operation from M1 can start on the cycle after the start of M2. Concretely, our ordering enforces a cycle distance between two consecutive executions of a single operation to be greater than the number of cycles between the start of the first and the start of the last operation within the iteration; if this value is higher than the initial loop II, it can constrain the throughput.

It is important to note that our ordering condition from Section 4.3.2 is *sufficient* to guarantee the absence of deadlock in any dataflow circuit with the structural properties described in Chapter 2. This condition is not always *necessary*—our generic ordering mechanism could be replaced by application-specific multiplexing and buffering schemes. For instance, one could relax the ordering constraint such that a particular number of executions from different iterations are allowed to overlap—in the example above, allowing an operation from M1 to start before the operation of M2 from the preceding iteration would lower the II.

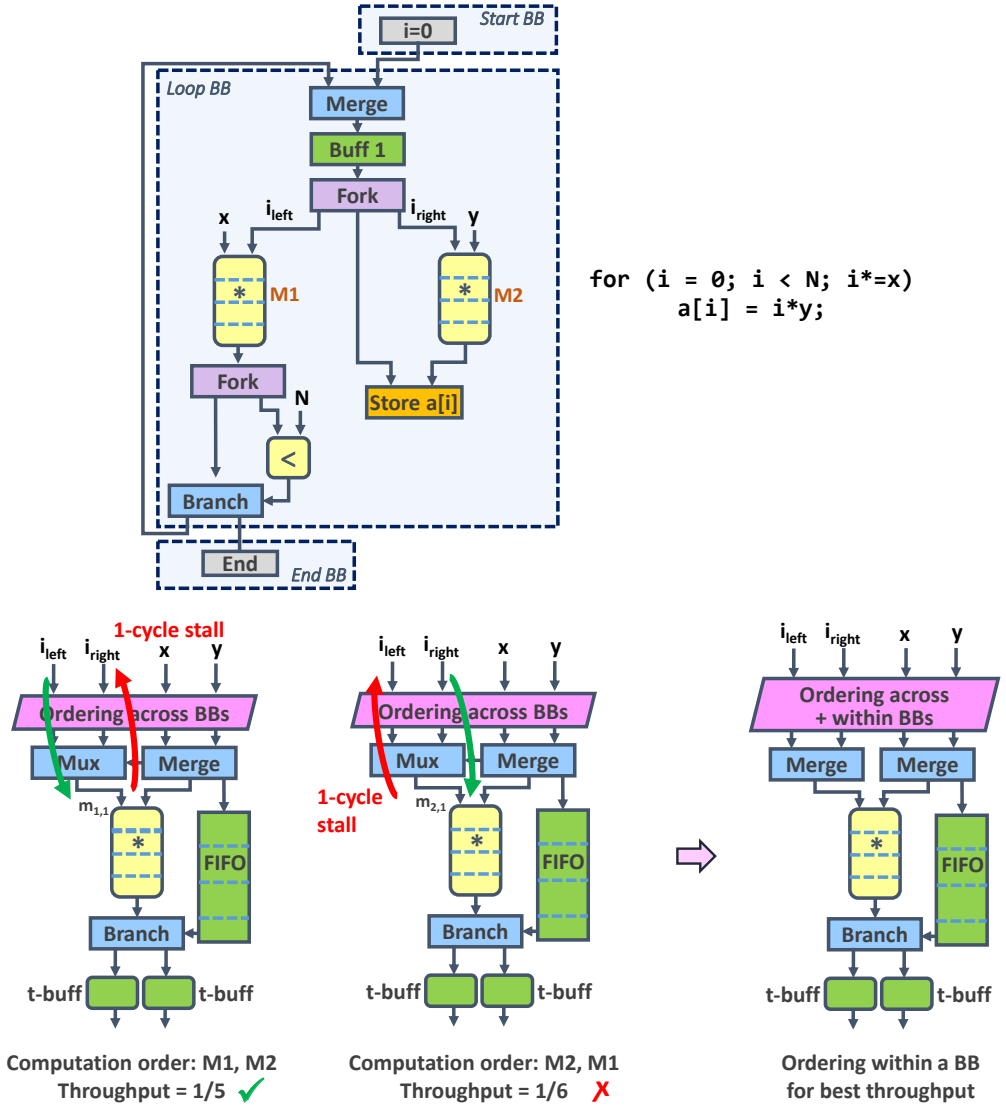


Figure 4.5 – Performance impact of sharing. The order in which tokens are sent to the shared unit may impact performance by postponing the execution of a throughput-limiting computation (in this example, M1 on the cyclic path). Hence, apart from ordering tokens from different BBs as they enter a shared unit, we also enforce the ordering of tokens of the same BB which has the minimal performance impact.

Intuitively, the number of overlapping iterations could be determined based on the cycle distances between operation executions and the relation of this value with the achievable II, as discussed above. To ensure the absence of deadlock, the buffers around the sharing logic would need to be sized to accommodate for tokens from multiple overlapping iterations. The technique from Section 4.3.3 would then be extended to order operations among all iterations which overlap. Naturally, the search space for the appropriate ordering would increase with the number of overlapping iterations and the ordering logic would grow with the more complex ordering constraints. Without loss of generality, we here limit our ordering to a single iteration and the rules from Sections 4.3.2 and 4.3.3. As we will later see, our ordering strategy effectively implements sharing without a throughput penalty in realistic benchmarks.

4.4 Ordering Implementation and Model

In the previous section, we showed that ordering tokens at the inputs of the shared unit ensures that the circuit is functional and that its original throughput is maintained. We here detail how to implement this ordering in the dataflow circuit as well as how to account for it when analyzing circuit performance.

4.4.1 Implementation

To implement the desired ordering between operations which share a unit, we build an additional network of dataflow units that strictly mimics the control flow of the program; this in-order network propagates a data-less token which triggers the advancement of operands to the shared unit in a predetermined order and only when control flow reaches the corresponding BB. Each shared operation is associated with a lazy fork in this network; this fork is synchronized (using a join) with a particular set of inputs to the unit. The fork must be lazy (see Section 2.3.1) so that a token moves forward in the network and triggers the next fork only after the joined inputs have been sent to the unit. The forks are separated by buffers which introduce a 1-cycle sequential delay, i.e., two forks cannot be active at the same time. Hence, only one set of inputs to the unit will be active at any given clock cycle and the order of activations corresponds to the desired operation ordering.

Figure 4.6a shows a dataflow circuit with two CFG cycles; the circuit contains three multiplications which we, in this example, aim to implement using a single multiplier (all other dataflow units are omitted from the figure for clarity). The in-order network which supplies ordering information to the shared unit is shown on the left of Figure 4.6b—in this example, it implements the orderings {M1, M2} in BB1 and {M3} in BB2. When the execution of BB1 starts, the first fork keeps the token until both inputs of M1 become available and are consumed by the multiplier; only then does the token from the lazy fork move to the next fork through a buffer, triggering the execution of M2 at least once clock cycle later. Similarly, assuming that the control flow decides on the execution of BB2, the in-order network will ensure that M3 executes before the multiplications from the next iteration of BB1 (i.e., M1 and M2). The in-order network of Figure 4.6b together with the joins effectively implements the functionality of the rightmost ordering component in Figure 4.5.

4.4.2 Sharing Model for Performance Analysis

To determine whether sharing affects performance, as discussed in Section 4.3.3, we need to analyze the throughput achieved in each CFG cycle and compare it to the throughput achieved prior to sharing. The latter is directly obtainable for the circuit in Figure 4.6a using the performance analysis from Chapter 3 which provides us with the throughput of the two CFG cycles. However, the circuit representation in Figure 4.6b is not directly suitable for performance analysis—determining the throughput of each CFG cycle requires every operation to be repre-

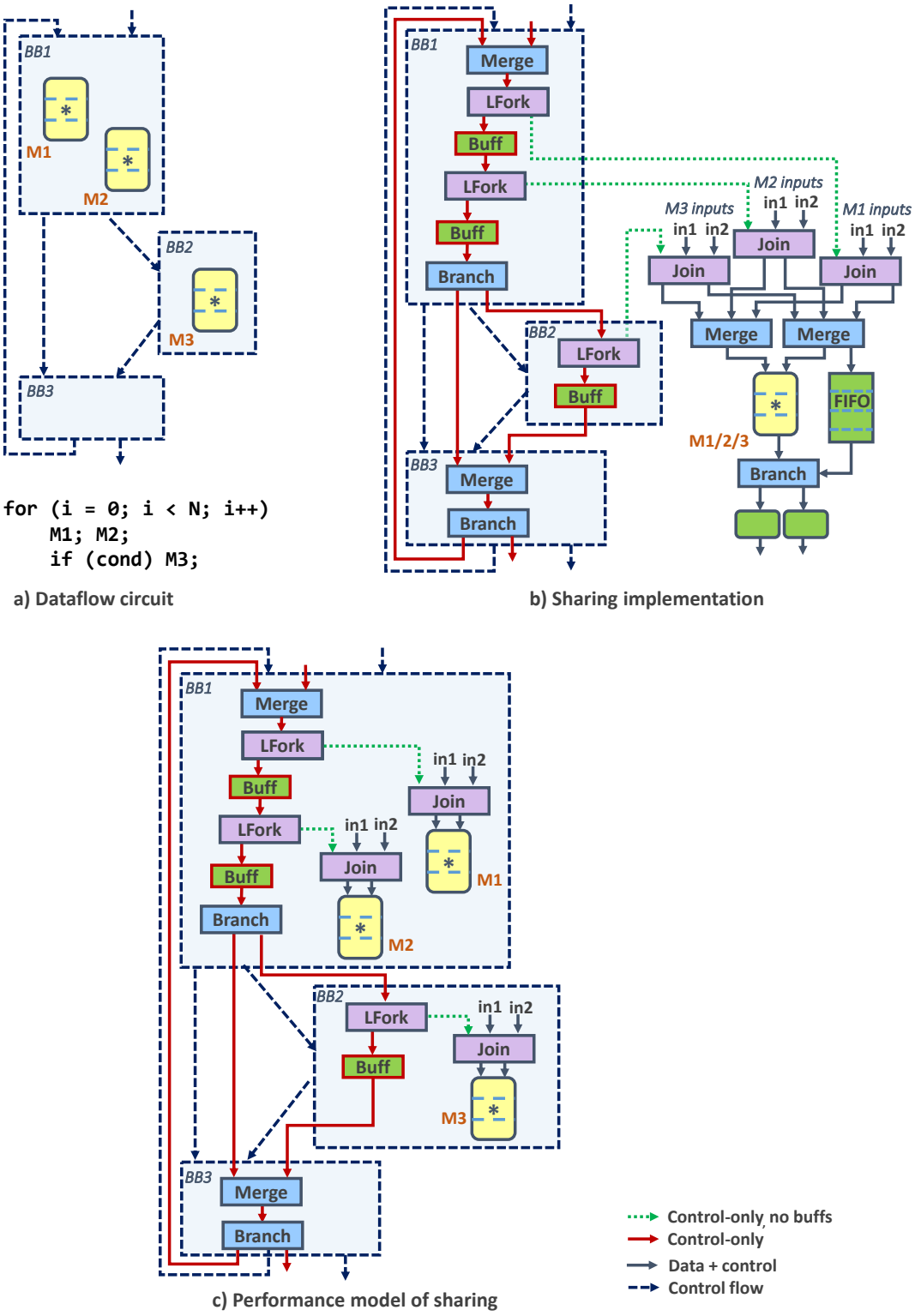


Figure 4.6 – Sharing implementation and model. A specialized in-order dataflow network enforces the specified ordering of operations in the shared unit. The same network is used during performance analysis to model delays introduced by sharing.

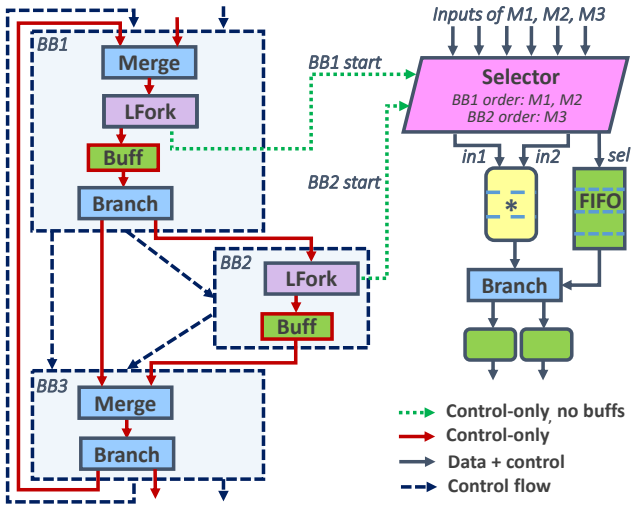


Figure 4.7 – Optimized sharing implementation. We use the in-order network described in Section 2.3.3 which, instead of sending an ordering signal per operand, sends a signal per BB; the selector uses this information to enforce a preencoded ordering of operations within each BB.

sented as an individual unit in a particular BB (and, consequently, analyzable as part of a CFG cycle); furthermore, the merge and branch units at the shared unit inputs and output are not immediately compatible with the choice-free behavior that such performance analysis requires, as detailed in Section 3.2.1.

Hence, for the performance analysis, we represent each operation *individually* in its original BB and model the effects of sharing with the in-order network described in the previous section; it connects the individual operators and describes the cycle-delays due to the enforced ordering, as shown in Figure 4.6c. The performance analysis will determine the throughput achievable with this circuit configuration and the corresponding delays. The comparison of the achieved throughput with that of the original circuit indicates whether the sharing and the explored ordering are desirable; we will include this aspect in the sharing strategy in Section 4.5.

4.4.3 Optimized Implementation

The sharing logic of Section 4.4.1 may quickly grow in complexity because each shared unit requires its own in-order network with as many lazy forks and buffers as there are shared operations; clearly, it is desirable to unify all these networks. Additionally, as we described in Section 2.3.3, our circuits already have an in-order network expressing the dynamic succession of BBs executed. Therefore, we adapt our implementation so that all shared units can directly leverage this existing network.

Our simplified implementation is shown in Figure 7. The network on the left of the figure is what already exists in the dataflow circuit: it emits tokens corresponding to the succession of BBs and, as our original network, the use of lazy forks separated by buffers ensures that each BB

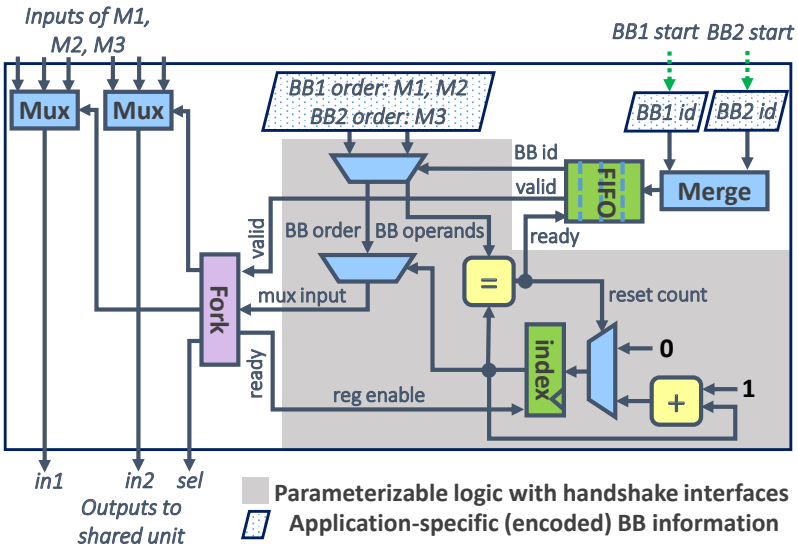


Figure 4.8 – Implementation of the selector unit. The internal selector logic (grey) selects the appropriate data inputs of the muxes on the left based on the order of BB execution (i.e., the *BB start* signals) and the preencoded operation order for each BB.

start signal is triggered strictly in order. Essentially, the difference compared to Figure 6b is that the *selector* receives a single ordering signal per BB instead of an ordering signal per operand: thus, every time a BB starts, the selector needs to enforce the ordering of the corresponding BB operands (preencoded in the selector unit) before the operands of the subsequent BB.

Figure 4.8 details the selector unit implementation. It contains a FIFO which stores the IDs of the incoming BBs as they arrive in program order and one at a time from the in-order network. The *BB id* at the head of the FIFO selects the preencoded ordering information of the corresponding BB (i.e., a vector with the operand order, *BB order*, and the total number of operands of this BB, *BB operands*). An internal counter enables the appropriate input ports (*mux input*) of the data muxes on the left of the figure; a mux port is enabled only after the previous port has sent a token into the unit. A *BB id* is removed from the queue when all its operations have started executing, moving the successor BB to the head of the queue and allowing its tokens to enter the shared unit next.

The size of the encoded ordering information depends on the total number of operations sharing the unit, S, maximal number of units within a single BB, S_{bb}, and the number of BBs connected to the selector, B: (1) *BB id* requires $max(1, \lceil \log_2(B) \rceil)$ bits, (2) *BB order* requires $S_{bb} \times max(1, \lceil \log_2(S) \rceil)$ bits, and (3) *BB operands* requires $max(1, \lceil \log_2(S_{bb}) \rceil)$ bits. Typically, only a few operations share a unit and these values are relatively small (e.g., in the example of Figure 4.8, *BB id*, *BB order*, and *BB operands* require 1, 4, and 1 bits, respectively). The complexity of the multiplexing logic in the selector follows the same trends; it is typically minor in comparison to the 32- or 64-bit data multiplexers (left of the selector in Figure 4.8), which are necessary in any sharing implementation and are not an overhead of our particular strategy.

4.5 Putting It All Together

In the previous sections, we discussed how to achieve correct and performance-efficient resource sharing in dataflow circuits; we here describe our sharing strategy, summarized in Algorithm 4.1.

Initially, every operation is considered a separate group (i.e., a separate unit). Our strategy attempts to merge different groups which can share the same physical resource without compromising the throughput of any of the loops as follows:

1. *Sharing within a loop nest*, i.e., within a strongly connected component of the CFG graph. For every pair of groups which belong to the same loop nest and whose sum of token occupancies is at most equal to the unit latency (i.e., which are determined underutilized, as described in Section 4.2), we exhaustively explore all ordering combinations between the operations of these groups until an ordering which does not damage the throughput of any of the loops is found. The ordering is modeled as illustrated in Section 4.4.2 and the throughput evaluation is based on the performance analysis from Chapter 3. If such an ordering exists, the groups will be merged and the occupancy of the group will be updated; otherwise, the merging is discarded. This process repeats until no further merging can be performed without a performance penalty. The final ordering within each group corresponds to that found in the last successful merge and the buffer placement and sizing to that determined in the last grouping.
2. *Sharing across loop nests.* In this step, we merge every distinct group of one loop nest with any distinct group of another (if available and not already merged with another group from the same loop nest); the ordering of operations of each BB remains as determined in the previous step.
3. *Sharing other units.* We merge the units which do not belong to any loop with any of the groups from the previous steps.

The first step ensures that sharing never damages the throughput of any of the interconnected loops. In the second step, throughput analysis is not needed because different loop nests, in general, execute consecutively—while some final iterations of one loop may overlap with the initial iterations of another, two operations from different loop nests will never execute simultaneously in the steady state. The same holds for units which do not belong to any loop.

Figure 4.9 illustrates this strategy with an example. The given CFG has two loop nests; although not visible in the figure, we assume for the purpose of this example that M1 and M3 are on paths which are critical for throughput and, hence, should not be shared. In the first step, the operations are grouped while evaluating throughput, as shown on the right of the figure. The second step groups operations across the loop nests (e.g., M5 is grouped with M1 and M2) and the third step adds the remaining operations (M4) to the previously determined groups. In this example, the resulting circuit implements five operations using two shared units.

Our strategy minimizes the number of units under a throughput constraint. It is possible to adapt it to other optimization objectives as well, e.g., honoring a resource constraint: if the constraint

```
// Input:  units (all units of the same type, e.g, mul, div,...)
// Input:  sets (CFG loop nests, i.e., strongly connected
// components of the CFG)
// Output:  globalGroups (sets of operations which share a resource)

// 1.  Sharing within a loop nest
forall s ∈ sets do
    // Initialize groups to individual units of the loop nest
    groups (s) = {u | u ∈ units, u ∈ s}
    // Grouping of units
    while groups (s) modified do
        forall g1, g2 ∈ groups (s), g1 ≠ g2 do
            // If sum of token occupancies is at most equal to
            // the unit latency, sharing may be possible
            if Θ̇_g1 + Θ̇_g2 ≤ L_u then
                // Exhaustive search for best ordering in group
                forall ord ∈ possible_orderings (g1 ∪ g2) do
                    // Use MILP to evaluate throughputs
                    if throughput (s, ord) = throughput (s) then
                        // Merge groups and update the ordering
                        groups(s).update (g1, g2, g1 ∪ g2, ord)

                        // Update token occupancy of merged group
                        Θ̇_g1∪g2 = Θ̇_g1 + Θ̇_g2

                        // Terminate ordering search
                        break

// 2.  Sharing across loop nests
globalGroups = {}
forall s ∈ sets do
    // Merge every distinct group of one loop nest
    // with distinct groups of other nests
    i = 0
    forall group ∈ groups (s) do
        globalGroups (i++).add (group (s))

// 3.  Sharing other units
// Merge every remaining unit with any of the existing groups
i = 0
forall u ∈ {u | u ∈ units, ∀s ∈ sets: u ∉ s} do
    globalGroups (i++ mod globalGroups.size).add (u)
```

Algorithm 4.1: Sharing strategy.

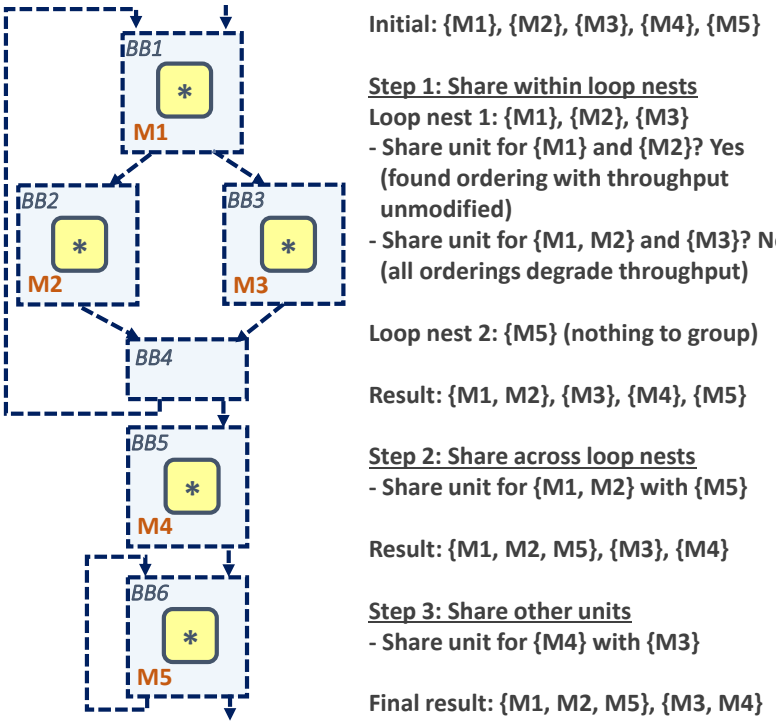


Figure 4.9 – Example execution of our sharing strategy. The execution assumes that units M1 and M3 are on throughput-critical cycles and, therefore, should not be shared.

is tighter than the group count achieved naturally by our strategy, one could continue grouping until it is met; the associated performance penalty could be minimized by exploring different groupings. Our algorithm immediately identifies good sharing candidates (i.e., underutilized units) for efficient exploration; the search space could be further reduced by a priori excluding certain orderings within particular groups. We note that providing an efficient algorithm is orthogonal to our main contribution: showing how to systematically build functional dataflow circuits which implement sharing.

4.6 Evaluation

In this section, we demonstrate the ability of our approach to implement resource sharing in dataflow circuits obtained from C code.

4.6.1 Methodology and Benchmarks

We use the methodology from Chapter 2 to implement our dataflow circuits and the performance optimization from Chapter 3 to maximize their throughput. We evaluate a selection of floating-point kernels from the PolyBench benchmark suite [98], which contain loop nests with different properties and computational patterns; most have long-latency loop-carried dependences due to pipelined floating-point operations which limit the loop initiation interval. In addition, we

Table 4.1 – Resources of dataflow circuits without sharing (i.e., *Naive*) and with sharing (i.e., *Shared*), after place-and-route with Vivado.

Bench-mark	DSPs			LUTs			FFs		
	Naive	Shared	ratio	Naive	Shared	ratio	Naive	Shared	ratio
atax	10	5	**0.50**	1970	2076	1.05	2206	1997	0.91
bicg	10	5	**0.50**	1627	1602	0.98	2018	1814	0.90
gemm	11	5	**0.45**	2339	2448	1.05	2500	2491	1.00
gemver	28	10	**0.36**	5580	5433	0.97	6753	5418	0.80
gesummv	18	5	**0.28**	2648	2666	1.01	3163	2528	0.80
2mm	16	5	**0.31**	3785	4200	1.11	4155	4153	1.00
3mm	15	5	**0.33**	3700	3653	0.99	3524	3096	0.88
mvt	10	5	**0.50**	2017	2029	1.01	2253	1878	0.83
if loop add	4	4	**1.00**	960	960	1.00	1318	1318	1.00

revisit the *If loop add* kernel from the previous chapter. Our primary goal is to minimize the DSP usage without affecting loop throughput; we hence attempt to share floating-point operations which are realized in DSPs following the strategy from Section 4.5 and implemented as described in Section 4.4.3.

We use ModelSim to measure execution cycle count and to verify functional correctness. Our designs target a Xilinx Kintex-7 FPGA and employ Xilinx floating-point arithmetic operations (encapsulated in custom wrappers with handshake signals to communicate with the other the dataflow units), whereas memory operations connect to dual-port BRAMs. We use Vivado to obtain the clock period and resource usage of our designs after placement and routing.

4.6.2 Results: Effectiveness of the Sharing Strategy

To evaluate our technique, we compare the resources and performance of dataflow circuits which do not implement sharing with the circuits optimized using our sharing strategy.

The results of our comparison are shown in Tables 4.1 and 4.2. The circuits without sharing (labeled as *Naive* in the tables) achieve the best possible pipelines (i.e., limited exclusively by the loop-carried dependences) and with a minimal number of cycles. However, they employ an individual functional unit for each operation in the code, which is reflected in their DSP usage. In contrast, the designs optimized with our strategy (labeled as *Shared*) share functional units among multiple operations of the same type and, consequently, significantly reduce the number of employed DSPs. The only example where this is not the case is *if loop add*, which implements a perfect pipeline, hence, sharing is not desirable—we will further discuss this situation in the following section. Our strategy ensures that the loop throughput remains unchanged; this is evident from the cycle count, which either remains identical to the naive solution or, in some cases, slightly increases. This increase is due to the increase in pipeline latency (i.e., some operations which use a shared unit execute later than in the original circuit, as described in Section 4.3.3) or transient effects when independent loops overlap (i.e., when one loop is ending and another one is starting, sharing might temporarily lower throughput while both loops

Table 4.2 – Timing of dataflow circuits without sharing (i.e., *Naive*) and with sharing (i.e., *Shared*). We measure the cycle count in simulation and obtain the clock period (CP) from Vivado, after place-and-route.

Bench-mark	Cycle count		CP (ns)		Exec. time (μs)		
	Naive	Shared	Naive	Shared	Naive	Shared	ratio
atax	4140	4459	4.9	4.3	20.3	19.2	0.95
bicg	7909	7910	4.6	4.3	36.4	34.0	0.93
gemm	68827	68827	5.7	4.9	392.3	337.3	0.86
gemver	1817	1899	5.1	5.6	9.3	10.6	1.15
gesummv	7952	8391	5.0	4.9	39.8	41.1	1.03
2mm	16610	17325	5.5	5.6	91.4	97.0	1.06
3mm	24557	24621	5.2	5.5	127.7	135.4	1.06
mvt	15708	15740	4.9	4.9	77.0	77.1	1.00
if loop add	1106	1106	5.0	5.0	5.5	5.5	1.00

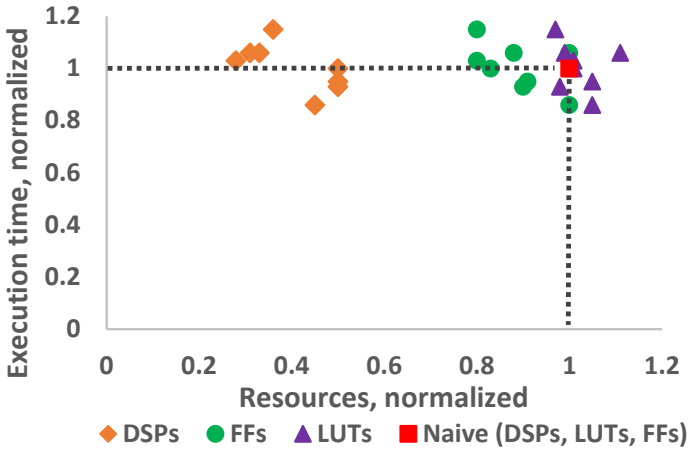


Figure 4.10 – Execution time and resources of dataflow circuits with sharing, normalized to the designs without sharing. Note that the main aim of our optimization strategy is to reduce the DSP count.

compete for a shared resource, as indicated in Section 4.5). These effects are perfectly in line with what we described earlier and arguably acceptable for the significant DSP savings.

The differences in clock period (labeled as CP in the table) are minor, indicating that sharing does not cause notable CP degradation; the differences are largely due to the timing variations caused by FPGA place-and-route. In addition to significant DSP reductions, our designs typically require fewer LUTs and FFs as well, which indicates that the complexity of the components that we introduce to implement sharing (i.e., selector at unit input, branch at unit output) is minor compared to the shared resources (i.e., shared computational units with the corresponding dataflow wrapper logic).

We summarize our main results in Figure 4.10, which shows the execution time (i.e., the product of the CP and the cycle count) and resources (i.e., DSPs, LUTs, and FFs) of our designs, normalized to the naive designs without resource sharing. It is important to note that all our solutions are Pareto optimal in terms of DSP units; while sharing never reduces the clock cycle count,

Table 4.3 – Resources of circuits produced by Vivado HLS (i.e., *Static*) and dataflow circuits with sharing (i.e., *Shared*). The *Shared* results are repeated from Table 4.1. The matching DSP counts indicate that our approach successfully identified all sharing opportunities. The LUT and FF overheads of dataflow circuits are completely expected, as the explored benchmarks are regular kernels which do not benefit from dynamic scheduling. The exception is *if loop add*, whose static version requires fewer DSPs, but at significantly higher execution time than the dynamic kernel.

Bench-mark	DSPs			LUTs			FFs		
	Static	Shared	ratio	Static	Shared	ratio	Static	Shared	ratio
atax	5	5	**1.00**	388	2076	5.35	762	1997	2.62
bicg	5	5	**1.00**	425	1602	3.77	824	1814	2.20
gemm	5	5	**1.00**	458	2448	5.34	837	2491	2.98
gemver	10	10	**1.00**	1032	5433	5.26	1631	5418	3.32
gesummv	5	5	**1.00**	553	2666	4.82	944	2528	2.68
2mm	5	5	**1.00**	598	4200	7.02	963	4153	4.31
3mm	5	5	**1.00**	666	3653	5.48	1104	3096	2.80
mvt	5	5	**1.00**	481	2029	4.22	802	1878	2.34
if loop add	2	4	**2.00**	315	960	3.05	525	1318	2.51

some designs even dominate their naive counterpart caused by the coincidental reduction in CP due to place-and-route. While we opted to identify sharing opportunities which do not affect throughput, our sharing mechanism can be easily extended to explore the design space and discover other Pareto optimal solutions.

4.6.3 Results: Comparison with Static HLS

In the previous section, we demonstrated that our methodology effectively shares units and reduces the resource requirements of dataflow designs. We are now interested in comparing the capabilities of our sharing strategy with that of a standard statically scheduled HLS tool. It should be noted upfront that, aside from *if loop add*, none of the benchmarks we here explore have characteristics which can take advantage of dynamic scheduling. Hence, it is reasonable to expect that our circuits incur resource (i.e., LUT and FF) and timing (i.e., achieved CP) overheads in comparison to those from a classic tool; we will evaluate and discuss these costs in detail in Chapter 9. Our purpose here is to investigate whether the unit count (i.e., number of employed DSPs) achieved by our sharing strategy matches that of state-of-the-art HLS solutions.

We synthesized the benchmarks from Section 4.6.1 with Vivado HLS; in all of them, we employ the pipeline directive in the innermost loops. We do not impose any resource constraints—hence, the tool shares as many units as possible without damaging the II, which qualitatively matches the strategy of our algorithm from Section 4.5 and makes the solutions directly comparable.

Tables 4.3 and 4.4 compare the results obtained by Vivado HLS with dataflow circuits which implement sharing (i.e., the *Shared* columns repeat the results from Tables 4.1 and 4.2). As indicated in the DSP column, the static designs employ the exact same number of DSPs as our solutions in Table 4.1, which validates that our sharing strategy successfully identified all sharing

Table 4.4 – Timing of circuits produced by Vivado HLS (i.e., *Static*) and dataflow circuits with sharing (i.e., *Shared*). The *Shared* results are repeated from Table 4.2. The CP overheads of dataflow circuits are completely expected, as the explored benchmarks are regular kernels which do not benefit from dynamic scheduling. The exception is *if loop add*, whose static version requires fewer DSPs, but at significantly higher execution time than the dynamic kernel.

Bench-mark	Cycle count		CP (ns)		Exec. time (μs)		
	Static	Shared	Static	Shared	Static	Shared	ratio
atax	5041	4459	3.3	4.3	16.6	19.2	1.15
bicg	9421	7910	3.3	4.3	31.1	34.0	1.09
gemm	91201	68827	3.2	4.9	291.8	337.3	1.16
gemver	2534	1899	3.4	5.6	8.6	10.6	1.23
gesummv	9029	8391	3.4	4.9	30.7	41.1	1.34
2mm	24402	17325	3.3	5.6	80.5	97.0	1.20
3mm	34803	24621	3.3	5.5	114.8	135.4	1.18
mvt	18782	15740	3.3	4.9	62.0	77.1	1.24
if loop add	10014	1106	3.2	5.0	32.0	5.5	0.17

opportunities in the benchmarks we explore. None of the benchmarks suffer from the fact that our technique enforces operation ordering across BBs, as described in Section 4.3.4, which indicates the effectiveness of our current approach in a large variety of practical cases. The only example where the DSP count is mismatched is *if loop add*, yet for a completely different reason: the if condition within the loop prevents the static HLS tool from pipelining and the conservative II makes sharing possible.

On the other hand, our solution for *if loop add* has a variable II which depends on the actual control outcomes and is, in the best case, equal to 1—the dynamic kernel does not share resources but it achieves a perfect pipeline and its execution time is, therefore, significantly lower than that of the static kernel. This example is representative of a classic tradeoff between static and dynamic scheduling [27]: the static kernel achieves minimal resources but suffers in execution time, whereas the dynamic kernel effectively pipelines the design for performance benefits and at a natural resource penalty.

As anticipated, the static kernels require fewer LUTs and FFs and achieve a lower CP (typically resulting in a lowered overall execution time) than their dynamic counterparts. Our goal here was to share computational resources (i.e., DSPs), which we have successfully achieved. Surprisingly, we note that all our solutions require fewer clock cycles to execute than the static solutions—while this effect is expected for *if loop add*, as discussed above, there is no fundamental reason for the dynamic kernels to execute faster in the other, perfectly regular, benchmarks. There are two explanations for this effect: (1) in some cases, our designs overlap different loops more effectively than Vivado HLS; although a similar overlapping effect could be achieved by employing the dataflow pragma in Vivado HLS, this optimization limits resource sharing between overlapping loops [118] and prevents us from comparing DSP-optimal pipelined designs, and (2) in some cases, the retiming algorithms of Vivado place an additional register on the critical loops and

increase the II by one clock cycle in comparison to our solutions; we employ a different register placement strategy [77] which does not identify the need for this register. Both of these effects are orthogonal to our contribution and have only a quantitative effect on the results; the matching DSP counts of the static and dynamic designs clearly indicate the effectiveness of our sharing approach.

4.7 Conclusions

Resource sharing is one of the key optimizations in high-level synthesis; if dataflow circuits are to compete with standard HLS, they need to be able to exploit this optimization opportunity. In this chapter, we presented a resource sharing methodology for dataflow circuits and we described a sharing mechanism which achieves correct, deadlock-free execution. In addition, we presented a method to identify sharing opportunities which do not compromise performance. On a set of benchmarks, we demonstrated the ability of our approach to significantly improve the resource efficiency of dataflow circuits. Our sharing mechanism achieves different area-performance tradeoffs in dataflow designs and makes them competitive in terms of computational resources (i.e., functional units and the corresponding DSP count) with circuits achieved using standard HLS techniques.

5 An Out-of-Order Load-Store Queue for Spatial Computing

Dataflow circuits achieved so far implement high-throughput pipelines and save resources through sharing; however, they still lack the mechanisms to support dynamic features which make them superior to statically scheduled circuits. Figure 5.1 shows a simple example, similar to the one in Figure 1.1a, where static memory access disambiguation is impossible; in this case, a statically scheduled accelerator would have to delay reading `a[b[i+1]]` until just after the write to `a[i]` of the previous iteration, conservatively assuming a dependence between the two accesses. On the other hand, a dynamically scheduled accelerator could potentially start a new iteration every cycle (in the absence of an address collision) and gain in performance.

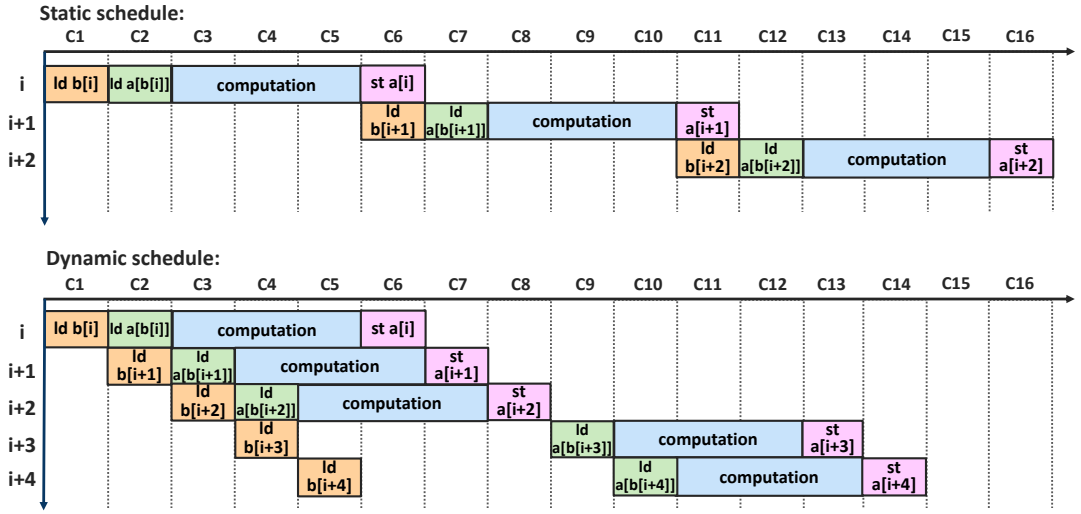
Supporting such dynamic behavior requires a memory interface that analyzes data dependences, reorders memory accesses, and stalls in the presence of effective data hazards. Although these features have been exploited in out-of-order processors for decades, there has been little effort to create generic accelerator-memory interfaces supporting out-of-order execution. The reason lies in a fundamental difference between the two systems: In a processor, the notions of fetching and decoding instructions immediately convey the *correct* (or, more precisely, *a* correct) sequential order of requests at the memory interface. In contrast, spatial circuits lack such notions and, in the construction of a dataflow-like accelerator, the information of the original sequential program order is lost unless explicitly maintained in an alternative manner.

In this chapter, we present an out-of-order load-store queue (LSQ) as an efficient interface between a dataflow accelerator and memory. We detail the construction of the LSQ and present a novel allocation policy which differentiates our LSQ from those found in standard processors.

5.1 Inadequacy of Processor Load-Store Queues

Consider a nonspeculative out-of-order processor with a traditional LSQ, similar to the one shown in Figure 5.2, at the memory interface. The LSQ entry depicted here is generic and

This chapter is based on the work published at the *International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, 2017 and the *ACM Transactions on Embedded Computing Systems*, 2017 [71].




```
for (i=0; i<N; i++) {
   a[i] = const*a[b[i]];
}
```

Figure 5.1 – A schedule created by an HLS tool unable to disambiguate dependences, compared to a dynamic schedule possible with a dataflow approach. The HLS tool will conservatively assume a dependence between every loop iteration, whereas the dynamic design stalls only in the presence of an actual dependence (in the figure, this is the case between iterations $i+2$ and $i+3$).

contains at least four fields: (1) an opcode indicating whether the operation is a load or a store, (2) a memory address to access, (3) the data to be written (only used by store instructions), and (4) a completion flag initialized to 0 and set to 1 when the LSQ has executed the operation. In this example, the LSQ is organized as a single circular buffer with *head* and *tail* pointers—other organizations are common but differences are irrelevant for this discussion.

Figure 5.3 depicts the six-step mechanism by which a processor gets load ("LD") or store ("ST") instructions executed, emphasizing the interaction with the LSQ [64]:

- *Instruction Fetch/Decode*. The processor fetches the instruction from the I-cache, decodes it, ascertains that it is a load or a store, and passes it to the LSQ.
- *Allocate*. The LSQ allocates a new entry for the instruction at the end of the queue and logically connects the present instruction with the new entry.
- *Supply Arguments*. As the processor pipeline executes other instructions, it will eventually determine the memory address and, for stores, the data value to be written. This information is supplied to the LSQ which writes the actual address and data (if needed) in the reserved entry of the queue.
- *Execute*: The LSQ executes the memory operation when *ready*—that is, whenever it is sure that the operation does not depend on any of the accesses coming *before* (i.e., closer to the head) in the queue and not yet executed. The LSQ may be able to execute the operation locally (e.g., a LD operation that reads the value of a ST operation ahead of it in

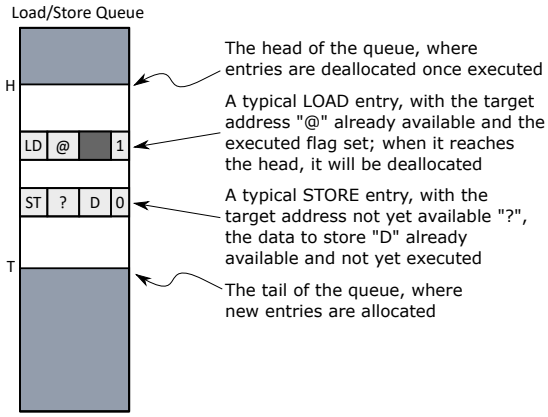


Figure 5.2 – A typical processor LSQ with head and tail pointers and two sample entries. In general, entries contain at least these elements: the operation type ("LD" or "ST"), the target address "@", the data to be stored "D", if appropriate, and a flag to signal completion. All addresses and data are usually not present at the moment of allocation (indicated with "?" in the figure).

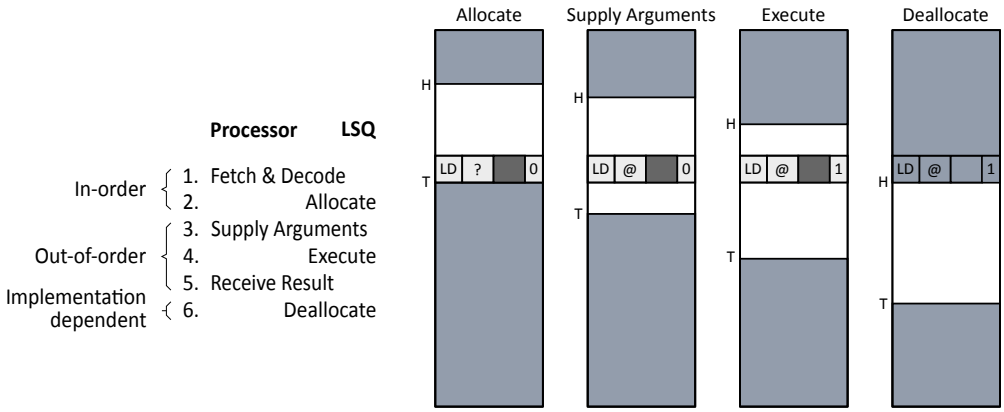


Figure 5.3 – The basic operation of an LSQ in an out-of-order processor. The allocation of entries in the queue must happen in sequential program order because the position in the queue is used to determine whether an access is safe to execute. Since, in a processor, this happens at *Decode* time, intrinsically in-order, everything is fine; unfortunately, spatial accelerators have no equivalent phase to *Decode*.

the LSQ), otherwise it transmits the operation to the memory subsystem. The LSQ sets the completion flag and returns the result to the processor.

- *Receive Result*: The processor receives the result of the memory operation from the LSQ (either a completion signal for store operations or a data value for loads).
- *Deallocate*: Eventually, the LSQ deallocates the entry for the instruction, freeing it for future use.

Steps 1 and 2 (Fetch/Decode and Allocate) *must* occur in the original sequential program order because this sequence implicitly specifies to the LSQ the potential dependences that need to be respected. In processors, the remaining steps typically occur out of order, but the LSQ ensures that the dynamically-arising memory dependences are correctly sequentialized.

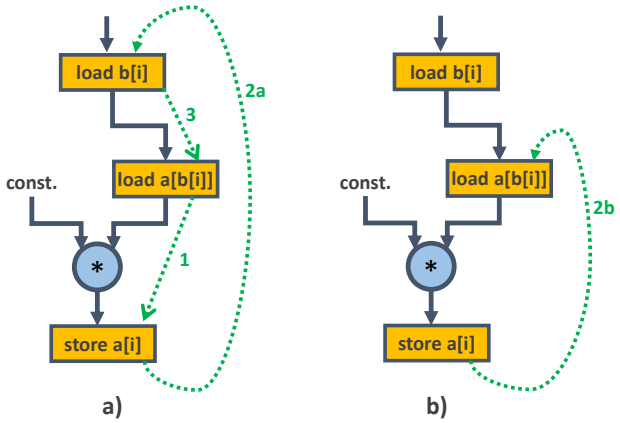


Figure 5.4 – A partial dataflow graph derived from the code of Figure 5.1. Nodes are operators (including memory accesses) and solid edges represent data flowing among them. Dashed edges correspond to the ordering of memory accesses in the original program. Static analysis in the compiler may transform or remove some of the edges (as shown in Figure 4b), but, generally, not all of them can be removed.

It is important to recognize that the LSQ shown in Figure 5.3 is built around the notion of a sequential program whose instructions are dynamically fetched and decoded depending on, among other things, the dynamic control flow of the program. The LSQ entries are allocated in the order in which instructions are fetched by the processor; this in-order allocation is critical because, once in the queue, potential dependences are verified between each entry and all preceding entries in the queue—and, if all dependences are satisfied, memory operations may execute out of order.

In a spatial computing system, there is no inherent notion of instructions or PC values. In terms of the described six-stage process, there is no Fetch and no Decode (Step 1); there are no instructions and no I-cache from which to read them. A typical spatial computing system is designed by transforming a dataflow graph into a circuit. In general, the dataflow graph obtained from the compiler will look like the example in Figure 5.4a, where solid edges represent actual data dependences and are transformed into physical wires in the circuit. Dashed edges indicate the sequence of memory operations (and, thus, the potential dependences) in the original program. The compiler may eliminate or transform some of these edges based on static analysis; however, it cannot safely remove dependence edges unless it is possible to guarantee, statically, that memory dependences cannot occur. In the example, edges 1 and 3 can be removed because they are implied by actual data dependences. On the other hand, edges 2a and 3 are unnecessary because static analysis may determine that the relevant accesses (to `a[]` and to `b[]`) can never conflict. Yet, the potential loop-carried dependence now needs to be represented by edge 2b, resulting in the graph of Figure 5.4b. Edge 2b cannot be eliminated because the two connected accesses lead to an incorrect result if reordered when their addresses collide.

We will discuss methods to analyze and optimize dependence edges between memory accesses in Chapter 6. The question here is: what to do with them when generating a dataflow circuit? Clearly, simply ignoring them would be incorrect, because then memory operations might

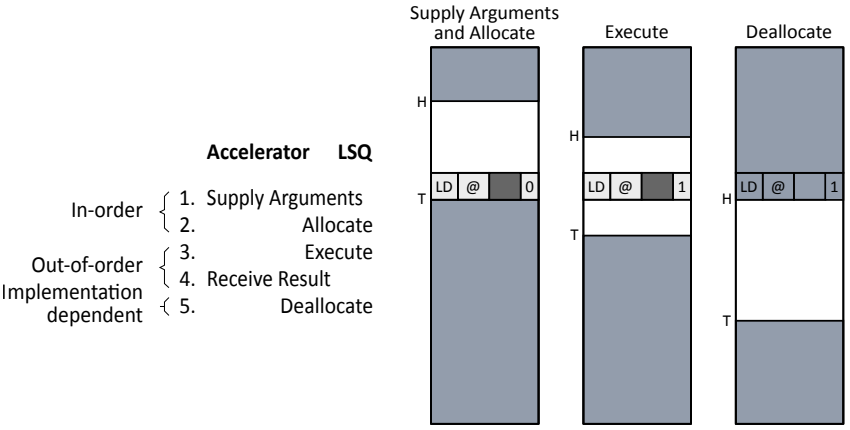


Figure 5.5 – Allocating entries when the arguments are supplied to the LSQ. This scheme can work correctly only if the circuits incorporate the dashed edges of Figure 5.4b: the arguments of a memory access must only reach the LSQ if all memory predecessors have supplied their arguments. The result is a major loss in the usefulness of dynamic dataflow execution and of the LSQ.

be triggered in any order and nothing in the circuit would carry the information necessary to respect these dashed edges when addresses collide. Prior research on dataflow computing has mentioned the possibilities of employing LSQs to resolve dynamic dependences (e.g., Huang et al. [66], among others), but has not described when one allocates LSQ entries in a spatial context—or, equivalently, by which mechanism one supplies ordering information to the LSQ. Answering this question is the foremost contribution of this chapter.

5.2 Supplying a Sequential Order to the LSQ

The allocation of entries to the LSQ must happen in *some* correct sequential order—i.e., an order that respects all dashed edges of Figure 5.4b, as this guarantees that all dependences are satisfied and that memory accesses are correctly executed. We here describe and contrast two design methodologies which ensure such behavior in dataflow circuits.

(1) *Allocating entries to the LSQ as their arguments arrive.* The interaction between the accelerator and the LSQ is shown in Figure 5.5 and the idea is that allocation happens when the address and/or data for a memory access are known. To ensure correctness, the accelerator must be aware of dependences and delay the transmission of arguments to the LSQ until preceding accesses (in terms of the dashed edges of Figure 5.4b) have already been allocated. This qualitatively corresponds to the scheduling approach of a classic HLS tool—the static schedule ensures that memory accesses are allocated in sequential program order. However, this approach nullifies the potential advantages of dynamic scheduling for applications that have ample memory parallelism that cannot be discovered statically. For example, such a tool could not produce the dynamic schedule shown at the bottom of Figure 5.1. Hence, the benefits of integrating such an LSQ into an accelerator are limited [9, 66].

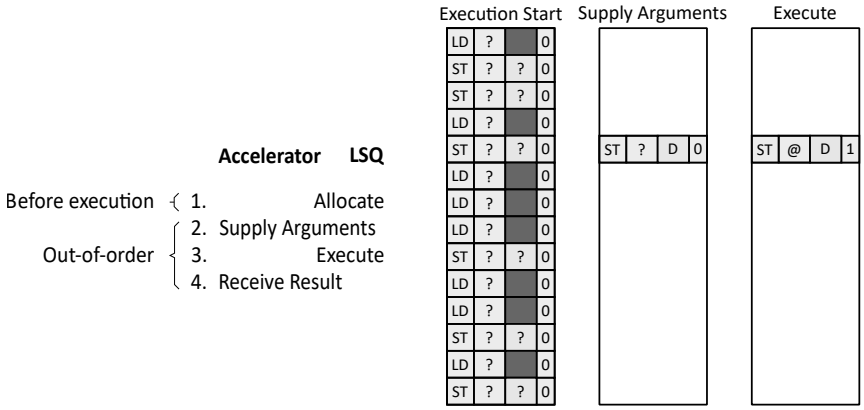


Figure 5.6 – Allocating entries statically before execution. This scheme depends on the possibility of determining all accesses and their sequential program order at compile time, limiting it to only the most trivial applications. Note that there is no more head and tail of the queue, since the queue contains exactly all memory accesses of the program.

(2) *Static allocation before execution.* As an alternative, the HLS tool could statically allocate all memory accesses through one sequence that respects all memory dependence edges (see Figure 5.6). Such a system would allow the accelerator to supply operands as soon as they are ready and the LSQ could issue memory accesses as soon as all dependences are resolved; this would simplify the design of the accelerator and achieve the highest possible memory performance. Although correct in theory, this idea is only feasible for trivial applications that feature statically determinable control flow—which, once again, excludes the example of Figure 5.1, unless `N` is a compile-time constant. This scheme is also impractical, as the LSQ would have as many entries as the number of static memory accesses in the application, which would be unrealistically large in all but the most trivial of cases. Others have shown that both the critical path (assuming single-cycle accesses) and resource requirement demands grow as a function of the number of LSQ entries [117]; our implementation results confirm this observation.

Figures 5.5 and 5.6 represent these two extremes: The former dynamically allocates LSQ entries when arguments arrive, but must supply them in a statically-determined order which complicates the control mechanism and limits the ability of the LSQ to dynamically resolve dependences and issue memory accesses in parallel. The design in Figure 5.6 makes perfect use of the LSQ and allocates each LSQ entry as soon as its arguments are ready, giving the LSQ flexibility in terms of dynamically disambiguating memory accesses. However, the static analysis required of the compiler is unrealistic for non-trivial applications; LSQ area utilization and performance would degrade significantly as a result of the large number of entries. We desire a policy that combines the flexibility of the former strategy with the effectiveness of the second.

5.3 Our Allocation Strategy

We desire a memory interface that can execute accesses out of order when dependences allow it. As in ordinary processors, ordering is expressed by organizing the accesses in a linear reference

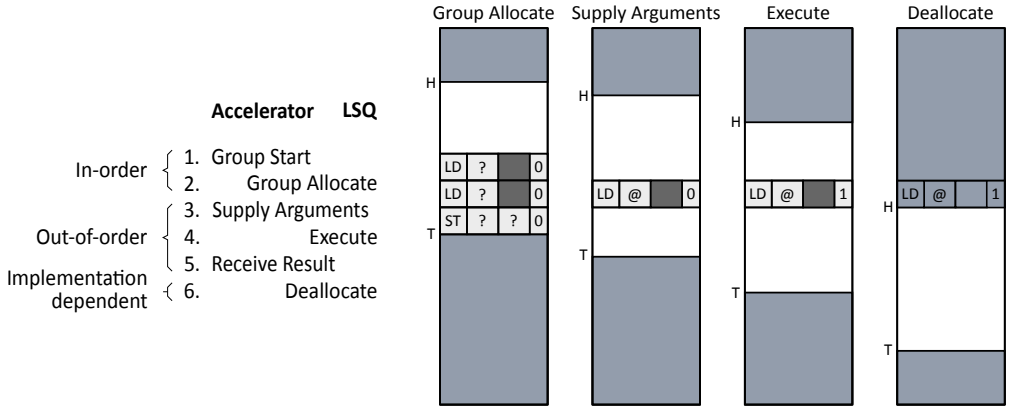


Figure 5.7 – Allocating entries by groups. Our solution requires the accelerator to announce groups of accesses when they become available. Groups are predefined sequences of accesses which are statically known to a compiler and that can be inserted atomically. Once this is done, our LSQ works essentially like a standard processor LSQ.

Table 5.1 – Comparison of different entry allocation options.

Strategy	Pros	Cons
Dynamically allocate entries once address or data are supplied to the LSQ	Applicable to any program	Accesses must be sequentialized to ensure correctness, LSQ is arguably almost useless
Statically allocate all entries before program execution	Fast out-of-order execution	Applicable only to programs with fully statically determinable accesses, unfeasible LSQ size
Dynamically allocate entries by group (our strategy)	Applicable to any program, fast out-of-order execution	

sequence (a queue): essentially, loads can be executed if the addresses of all preceding store entries in the queue are known and do not alias, and vice versa. As indicated in the previous section, the critical difficulty for spatial computing consists of creating such a sequence in the queue, because we cannot rely on the allocation policy of processors (in-order allocation at decode time). We thus propose a design option that is intermediary between the two approaches described in Section 5.2 and that circumvents their fundamental limitations.

To this end, we introduce the notion of groups: A *group* is a sequence of accesses which cannot be interrupted by a control flow decision (that is, if one access of a group executes, all other accesses belonging to the same group will eventually execute as well). Determining a correct order (i.e., an order that satisfies all the dependence edges) of accesses within a group is trivial using static analysis, as there is no control flow decision that could invalidate it. Our strategy is to dynamically allocate entries for all the memory operations of a group at once as soon as a control flow decision is made (as shown in Figure 5.7). Once the entries have been allocated, the

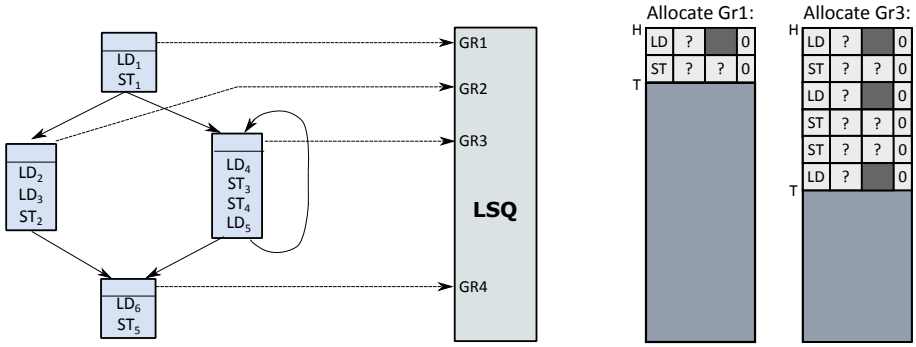


Figure 5.8 – A program with memory accesses divided into groups which are connected to the LSQ. Each group has a GR signal to indicate its start and to trigger the allocation of its memory accesses into the LSQ. In this case, groups correspond to basic blocks created by HLS tools. The figures on the right show two consecutive allocations (of group 1 and group 3) into the LSQ.

memory access arguments can arrive out of order and will be executed as soon as dependences can be determined (using the same strategy as a standard processor LSQ and described in Section 5.1). Table 5.1 outlines the pros and cons of the different allocation strategies.

Our group-based allocation principle can be summarized as follows: (1) The order of memory accesses of each group is produced through static analysis and encoded into the LSQ at compile time. (2) The LSQ has as many load/store ports as there are load/store operations in the program. (3) The ports are clustered by groups; every access port must belong to exactly one group. (4) Whenever the accelerator 'activates' a group, all load/store operations belonging to that group are allocated in the LSQ in the sequence that was statically determined for that group. (5) Once a group has been allocated, the LSQ expects each of the corresponding ports to get an access, eventually; dependences will be resolved based on the order of entries in the LSQ.

Consider a program with a control flow graph as shown in Figure 5.8 and with memory accesses logically divided into four groups, based on the control flow of the program. The order of accesses within each group can be statically predetermined using compiler analysis (we provide an arbitrary ordering for each group in the figure). Every group is connected to the LSQ with a dedicated GR signal which indicates to the LSQ the start of the particular group and triggers the allocation of its accesses into the LSQ. The queues on the right of Figure 5.8 show the allocation of group 1 and group 3 to the LSQ (assuming that a control flow decision determined the execution of group 3 after group 1).

The LSQ proposed here is designed to be independent of the synthesis algorithms that form groups and produce GR signals. Intuitively, each basic block could correspond to a group—we apply this strategy in the example in Figure 5.8 and in our dataflow circuits. We will detail our method to generate the GR signals in Section 5.5.

Our LSQ differs from traditional usage in processors for one key reason: even if an accelerator obtained from a program containing N memory operations does indeed require N memory

access ports, those ports seldom need to be connected to the same LSQ. Only accesses with the potential to conflict demand to share the same LSQ. In the example of Figure 5.1, if one determines that accesses to `a[]` and to `b[]` cannot conflict, one would need an LSQ with one load and one store port for `a[]` and a simple memory port (without an LSQ) to access `b[]`. Intuitively, using many small queues is beneficial for timing and area compared to a large queue, because both complexity and timing are certainly superlinear in the number of entries.

5.4 LSQ Implementation

Any LSQ needs to implement the following functionalities: (1) Allocate entries in the queue for new accesses. (2) Enable the access ports and connect them logically to the respective LSQ entries allocated in the previous step so that arguments and results can be dispatched. (3) Accept arguments for the allocated LSQ entries as they arrive out of order. (4) Dynamically decide which accesses can be safely executed without violating dependences in memory. (5) Return as soon as possible available results to the load ports that requested the accesses. (6) Deallocate entries in the queue when no longer needed. As pointed out earlier, almost everything related to steps (2) to (6) is identical or very similar to what happens in a traditional processor LSQ, whereas the implementation of function (1) is specific for spatial architectures.

As in any LSQ design, the challenge lies in implementing these functions concurrently with minimal complexity and achieving the lowest possible cycle time. For simplicity, we perform every function in a single cycle: this may result in a higher cycle time but has the significant advantage of naturally making every step atomic (i.e., starting on a rising edge and completing before the next edge), which avoids any difficulty with the concurrence of the operations (a function cannot change the state inconsistently while the others are executing as all start from a consistent state and all produce during a cycle their contribution to a new consistent state). Although a long cycle time could be severely damaging, we will show that even under this constraint the cycle time remains fairly affordable (that is, comparable to the cycle time of simple accelerators). Removing this constraint would result in a more complex design but would not significantly change the architecture and methodology we describe.

The only aspect where a multicycle nature is essential is the interface to the memory subsystem (caches, etc.): although our current implementation assumes that memory can be read in a single cycle, because this is the case in our system, it can be easily extended to support caches and complex memory hierarchies in future work. It is important to realize that accommodating this aspect of multicycle execution is much easier than the more general multicycle operation discussed above: the execution decision is dependent on the state of the queues and nothing in the system can revert this decision (that is, if a load can be executed before all other pending stores, no further event may invalidate such a decision). Thus, there are no issues of atomicity or concurrent state update here—the fact that the memory hierarchy provides a load result only after a variable number of cycles has virtually no influence on the design and does not present any of the difficulties that other multicycle updates may imply.

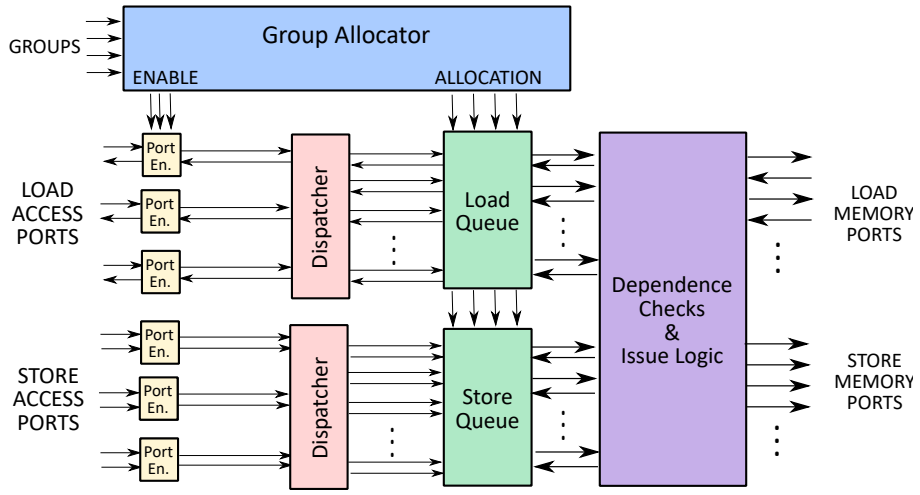


Figure 5.9 – Overall structure of the LSQ. At the center are separate load and store queues; the group allocator prepares entries in the queues and links them to the access ports; dispatchers connect the entries to the ports; the issue logic checks dependences and decides which entries are safe to send to the memory subsystem.

In the following sections, we discuss the overall structure of the LSQ and detail the six previously-mentioned functions, with particular attention to the allocation by group, which is unique to our architecture.

5.4.1 The Queues and the Overall Structure

The overall structure of the LSQ is shown in Figure 5.9. The logic around store and load entries of the queue is quite different, so we chose to implement two separate queues. Both queues have a head and a tail register and contain a power-of-two number of entries.

Each entry of the store queue contains the following fields: (1) store address, (2) store data, (3) number of the originating access port, (4) position in the load queue of the last load preceding this store (indicating which loads need to be checked for conflicts before issuing this store to memory—this mechanism will be discussed in detail in Section 5.4.2), (5) address validity flag, indicating whether the access port has already supplied the corresponding argument, (6) data validity flag, indicating if the access port has already supplied the data, (7) “executed” flag, indicating that the store has been issued to memory.

Each entry of the load queue contains the following fields: (1) load address, (2) a field for data received from memory, (3) number of the originating access port, (4) position in the store queue of the last store preceding this load, (5) address validity flag, indicating if the access port has already supplied the corresponding argument, (6) data validity flag, indicating whether the data has been received from memory, (7) “executed” flag, indicating that the load request has been issued to memory, (8) a flag indicating that the result has been sent back to the corresponding port. Figure 5.10 visualizes a load queue entry.

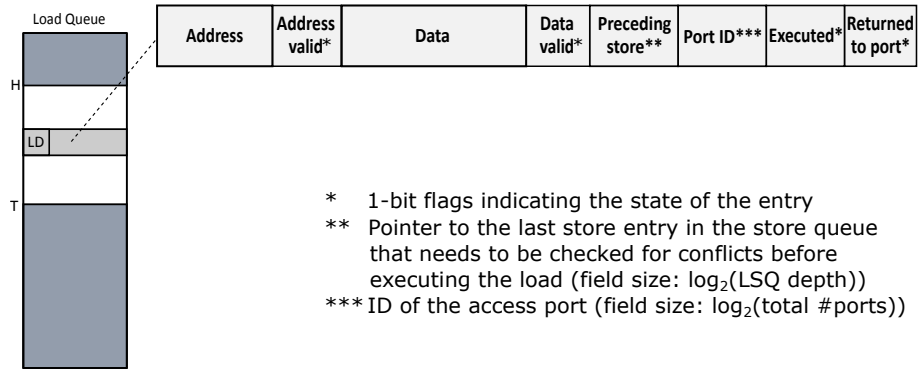


Figure 5.10 – Detailed load queue entry of the LSQ. Apart from the address and data fields, each queue entry contains the information about the port associated to the entry, the last preceding store request in the sequence, and multiple flags describing the state of the entry. The entries of the store queues are identical, apart from the last flag which can be omitted.

Despite the apparent complexity of the LSQ, as shown in Figure 5.9, concurrency is relatively straightforward since every element of the state of the LSQ is produced independently. For instance, the tails of both queues are only updated by the group allocator when new entries are added to the LSQ. The group allocator is also responsible for filling in the field pointing to the last preceding store/load position for each entry of the load/store queue. The access ports themselves, through the dispatchers, are the only components affecting the argument entries of both lists (memory addresses and store data). The parallel dependence checks and issue logic on the right side of Figure 5.9 are the only entities that can modify the executed flags and update the head pointers.

5.4.2 Group Allocator

The group allocator (Figure 5.11) is unique for our memory interface. When a group begins to execute, the first step is to allocate locations in the LSQ for the group's loads and stores, while ensuring that the allocation process respects their sequential program order, which is known statically. This enables the corresponding access ports (on the left of Figure 5.9) to accept memory requests and allows them to arrive in arbitrary order, while appropriately maintaining a mapping between memory operations and their LSQ entries.

Group allocation requests are sequential by definition, and thus only one new request can arrive at any moment in time. Once a group has been allocated to the queue, its accesses can execute before or in parallel with those in the other groups preceding it in the queue. Each group allocation request addresses a ROM at the core of the group allocator. This ROM outputs the following information: (1) number of loads in the group, (2) number of stores in the group, (3) for each load in the reference sequence, the number of stores preceding it in the group (we refer to this value as the offset value of the entry), (4) for each load in the reference sequence, the access port ID, (5) for each store in the reference sequence, the number of loads preceding it in the group (i.e., offset), and (6) for each store in the reference sequence, the access port

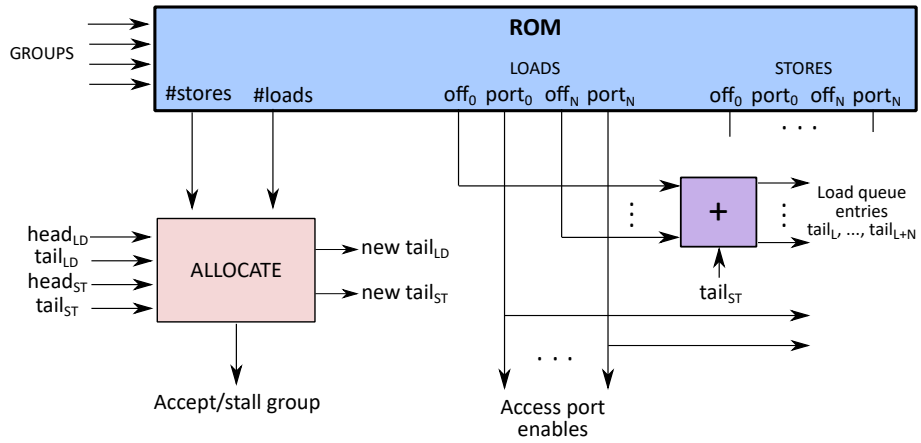


Figure 5.11 – Group allocator. When a statically-determined sequence of accesses (a group) begins, this unit allocates in parallel all the corresponding entries into each of the queues.

ID. For instance, group 3 from Figure 5.8 contains two loads and two stores whose sequential program order is load, store, store, load and these come from access ports *LD4*, *ST3*, *ST4*, *LD5*; the output of the ROM for this group is 2, 2, 0, 4, 1, 3, 1, 4, 2, 5. The first two values correspond to the numbers of the loads and the stores in the group, respectively, and sets (0,4), (1,3), (1,4), (2,5) represent (offset, port ID) pairs for each of the load and store ports of the group.

Using the first two output values of the ROM, the group allocator checks whether there is enough space in the two queues (in this example, both the load and the store queue need to contain two empty slots). If not, the allocation is deferred until the queues have enough space for the new group and no further allocations are accepted. If the allocation can take place, the group allocator tags each store by adding the number of preceding loads in the group (defined in the ROM) to the value of the load tail prior to allocation. This sum is a pointer to the position that the last preceding load occupies in the load queue (meaning that everything before it in the queue comes earlier in program order and needs to be checked for conflicts before issuing the store in question). This tag can correspond either to the previous load in the current group or to the last load of the previous group (in case the value provided by the ROM was zero). The tagging is done in parallel for each of the stores (and vice versa for each of the loads). Additionally, the allocator stores the ID of the originating port for each of the entries, which enables inserting values into the correct queue positions, and, for loads, returning the data fetched from memory to the appropriate port. Figure 5.12 shows the allocation process of group 1 and group 3 from the example in Figure 5.8, detailing the calculation of the tags of each queue entry.

The width of the ROM is determined by the size of the group with the largest number of memory accesses (e.g., group 3 in the case of the program in Figure 5.8, discussed above). It is worth noticing that, although in principle fairly wide, this ROM remains a small component in practical cases. In the small kernels that we evaluate here, the number of groups is quite small, noting that only accesses that cannot be statically disambiguated must be issued to the same queue. In the histogram kernel reported in Section 5.6, the ROM contains only a single 6-bit words.

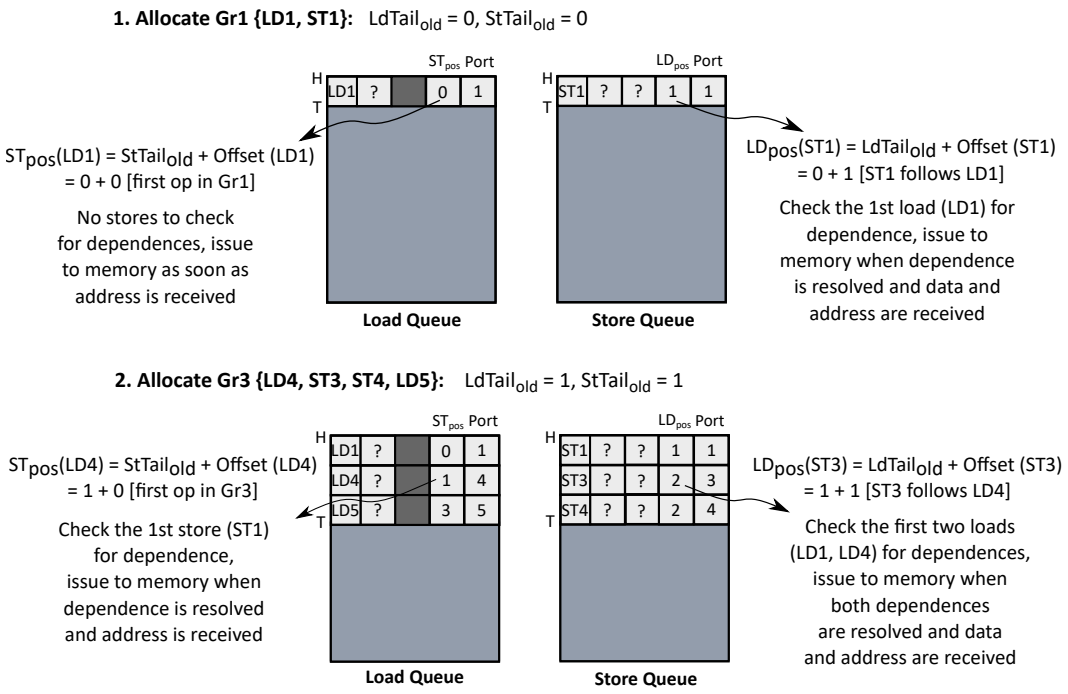


Figure 5.12 – Allocating groups to the LSQ. The figure details the allocation of group 1 and group 3 from Figure 5.8. The offset values and port IDs are obtained from the ROM. The group allocator calculates the last preceding load/store of each instruction, which the queue will use to perform dependence checks.

The critical path of the group allocator is fairly short and is essentially composed of the small ROM, an adder whose number of bits is sufficient to address the elements in the queues (to compute the absolute offset to save in the queues), and a multiplexer to bring the right offset to the corresponding queue entry.

5.4.3 Access Port Enable and Dispatchers

An access port is enabled by the group allocator once the allocator assigns queue entries for the group that this port belongs to. If the port sends a request prior to the allocation, it will be stalled until the corresponding position in the queue has been allocated. On the other hand, a group may be triggered again before all accesses of previous instances of the same group have received the operands and have executed (think of many iterations of a loop executing concurrently); this implies that ports may be linked to multiple entries of the queue. Therefore, the port has a counter to determine how many arguments it can accept; every time the corresponding group is allocated, the counter is incremented and every time the port receives an argument, the counter is decremented. The port is disabled (that is, it stalls arriving arguments) when the counter is null. Store ports contain separate enable logic for the data and the address, since they might not arrive simultaneously to the LSQ.

All ports search concurrently all elements of the corresponding queue for the earliest entry they are related to. The result of this search is used by the access port both to forward to the queue

newly arrived arguments and to pull newly available load results. The prioritization of queue entries related to the same access port acknowledges the fact that accesses on a specific port arrive in order and, most importantly, are returned in order. The critical path of the dispatchers is essentially a comparator for the port numbers, a priority encoder, and either the control path of a large multiplexer (with as many inputs as the elements of each of the queues, for returning load results) or the decoder logic of the same size (for writing arguments into the queue).

5.4.4 Checking Dependences and Executing

The rest of the LSQ is practically identical to any processor LSQ: dependences must be checked and accesses issued.

The load queue checks concurrently all loads, comparing each available address with all store addresses which precede it (and determined thanks to the pointer to the last preceding store that is part of the queue entry). If any of the preceding stores misses the address, the load is left waiting. If all preceding stores have an address and there is no collision (the load address is different from all store addresses), the load can be executed (i.e., sent to the memory subsystem). A priority encoder takes the oldest executable loads and passes as many as possible to the memory subsystem (i.e., as many as there exist ports to memory). If the load address equals one or more of the store addresses, and if the latest of the colliding stores has already received the data argument, the store is bypassed and the store data used as the result of the load access. The load result (either received from memory or bypassed from the store queue) is entered in the appropriate field of the load queue. As mentioned, load access ports pull concurrently all new results from the load queue.

The store queue acts similarly but only checks as many oldest stores (i.e., at the head of the queue and not yet executed) as there are store ports in the memory subsystem. For each, it compares the address, if available, with all load addresses which precede it (and determined thanks to the pointer to the last preceding load that is part of the queue entry). A store is executed only if (1) the address and the data for the store are known, (2) all preceding stores have executed, (3) the addresses of all preceding loads are known, and (4) there is no collision with any of the previous loads. If any of the tests fail, the store is kept waiting.

Executed accesses at the heads of the queues are simply deallocated. The queues can simultaneously deallocate as many entries as can be sent to the memory subsystem.

5.5 Connecting the Dataflow Circuit to the LSQ

Our allocation policy relies on the in-order generation of the GR signals for the LSQ; we here detail how to correctly produce and issue these signals from the dataflow circuit.

As mentioned earlier, we group memory accesses based on the BB they belong to (i.e., a single BB corresponds to a single group). Our challenge here is to guarantee that the signals coming

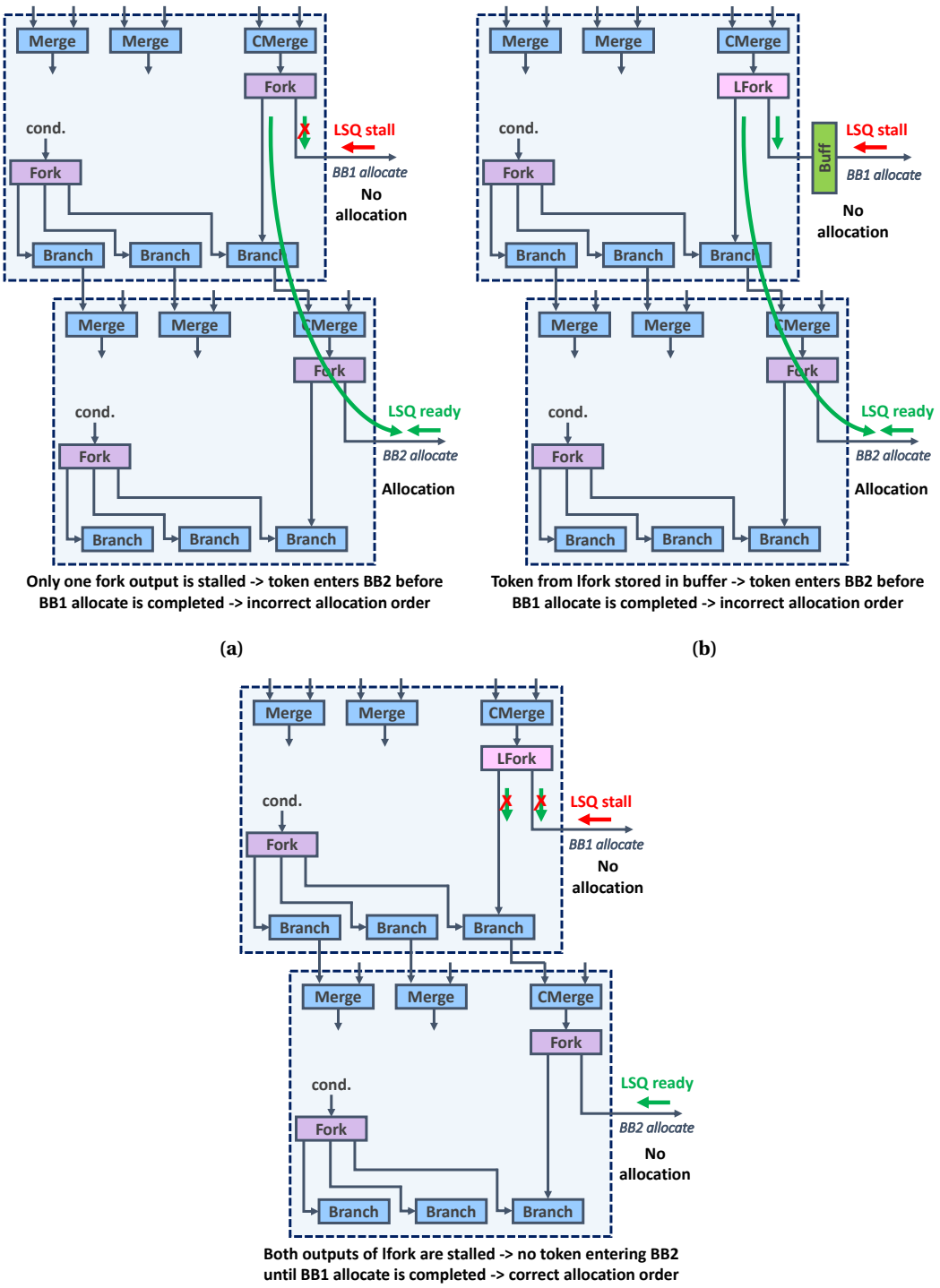


Figure 5.13 – Connecting the dataflow circuit to the memory interface. Figures 5.13a and 5.13b give examples of incorrect connections. In Figure 5.13a, the eager fork may send an allocation to BB2 before the allocation of BB1 completes. In Figure 5.13b, the allocation order may be reversed due to the storage element on the control line between the circuit and the LSQ. Figure 5.13c shows the correct way to connect the LSQ—an allocation cannot occur unless all predecessor allocations have been completed.

Table 5.2 – LSQ clock period (CP) and resource utilization for different numbers of groups.

Groups	Depth	Ports	CP (ns)	Slice	LUT	FF
1	8	8	4.6	1000	3498	1212
2	8	8	4.8	1243	4441	1287
4	8	8	4.9	1523	5541	1328

from the BBs needed for the LSQ are produced *in order* by a circuit which we have otherwise designed to be as aggressively out-of-order as we could. To this end, we exploit the in-order control path which we introduced in Section 2.3.3. The tokens in this path trigger the allocation of BBs as soon as the control flows there (i.e., as soon as a decision has been made to enter a particular BB). However, applying the standard dataflow circuit design strategy described in the previous chapters might result in the incorrect order of token arrival to the LSQ. Figure 5.13 shows two example situations leading to a potentially wrong execution: (1) If the token is forked to the LSQ using the typical eager fork, one of the fork outputs might send a token to the next BB before the LSQ has accepted a token from its predecessor (Figure 5.13a). (2) Although placing buffers in dataflow circuits has no impact on correctness (as discussed in Section 3.1), a buffer on the fork output connected to the LSQ might compromise the order of token arrival to the queue—if the token remains stored in the buffer, the successor BB could send a new token before the prior allocation has been completed (Figure 5.13b).

The correct way to connect the LSQ to the dataflow circuit is shown in Figure 5.13c: (1) The forks used to send the tokens to the LSQ are lazy forks (lforks)—if one of the fork outputs is stalled, the others will stall as well. (2) No sequential elements (i.e., buffers) are allowed on the fork outputs connected to the LSQ. This strategy ensures that a token can be passed to the successor BB only when the allocation of its predecessor BB has been completed—if an allocation is deferred (e.g., due to limited space in the LSQ), the token stalls and no further allocation requests reach the LSQ. This mechanism is exactly what we described in our implementation of resource sharing in Section 4.4.3, where we used it to convey the order of triggered BBs to the selector unit.

To connect our datapaths to memory, we leverage compiler analysis to simplify our memory interface—we will detail our approach in the following chapter. Whenever the compiler can disambiguate memory accesses, groups of accesses that cannot mutually conflict use separate LSQs, while accesses which cannot have dependences with any other accesses are connected to simple memory interfaces.

5.6 Evaluation

In this section, we discuss the resource and timing characteristics of our LSQ and evaluate how different queue parameters affect its performance and resource utilization. We demonstrate the

Table 5.3 – LSQ clock period (CP) and resource utilization for different numbers of ports.

Groups	Depth	Ports	CP (ns)	Slice	LUT	FF
1	8	2	4.6	1073	3655	1254
1	8	4	4.4	1049	3693	1249
1	8	6	4.5	1381	4866	1316
1	8	8	4.6	1000	3498	1212

benefits that the load-store queue brings by comparing our designs of applications containing irregular loops with those created by a standard HLS tool. All timing and resource information for our VHDL designs are from the post-routing analysis of Vivado. We provide the resource usage as the number of CLB slices, with the corresponding LUT and FF count.

5.6.1 Resource Utilization and Timing Analysis

Our queue designs are generated based on the required parameters (queue sizes, number of groups, number of ports, etc.). We here evaluate and discuss the timing and resource requirements when these parameters change.

Sensitivity to group count. We discuss the effect of the number of groups connected to the LSQ on the resources and the timing of the design. Table 5.2 provides the evaluation for queues with a varying number of groups, while keeping the overall number of load and store ports (distributed equally over the existing groups) and the queue depth constant. The change in the number of groups and the way the ports are organized impacts only the group allocator and has no effect on the rest of the design. Due to the parallel nature of this unit, changing its parameters has barely any influence on the cycle time, which is only slightly affected by the explored modifications. The resource requirements increase only moderately with the number of groups. This shows that our approach can effectively implement designs with different numbers of groups and implies the applicability of our solution to a wide range of applications.

Sensitivity to port count. In Table 5.3, we explore the effect of the number of ports on the resource and timing characteristics of our LSQ by comparing designs of equal queue depths and group count but containing different numbers of ports. We implement designs with an equal number of load and store ports (i.e., the 2-port design has one load, and one store port; we add one port of each type in every subsequent design). The number of ports does not significantly impact the cycle time. Although each port adds resources to the design, it can be noted that the overhead of a single port is minor compared to the overall resources of the LSQ. In some cases, the logic synthesizer can reduce the complexity of the logic around the queues thanks to the characteristics of the specific application it is customized for. This happens, for instance, for the 8-port LSQ in Table 5.3: In this case, because of the integer ratio between the queue

Table 5.4 – LSQ clock period (CP) and resource utilization for different queue depths.

Groups	Depth	Ports	CP (ns)	Slice	LUT	FF
1	2	2	3.6	102	254	286
1	4	2	3.7	354	1088	593
1	8	2	4.6	1073	3655	1254
1	16	2	5.9	4485	15922	2788

depth (8) and the length of the only possible group (4 loads, 4 stores), the LD/ST ports need to be connected only to specific queue entries. The logic synthesizer can use this information to reduce the design complexity and the effect is counterintuitive.

Sensitivity to LSQ depth. We compare now the timing and resource requirements for different queue depths, always with a single group and a fixed number of ports (one load, one store port). The results in Table 5.4 show a nonnegligible increase in resources and cycle time. Although our design exhibits ample parallelism and performs most operations concurrently, some functionalities cannot be implemented in constant time—for instance, to bypass data from the store to the load queue, one needs to check the store queue from the head to the tail to find the last conflicting data. This sensitivity to the number of queue entries is in line with the results reported by others in conventional LSQ designs—previous efforts to implement conventional LSQs in FPGAs have exhibited the same trends of resource and clock degradation with queue size [117]. These results motivate us to consider alternative design options in the future—our group allocation policy is generally applicable and can be incorporated into different queue architectures. It should be noted that our contribution here is not to improve the design of a conventional LSQ, but to propose a practical and efficient way to adapt conventional LSQs for spatial computing. As we will demonstrate in the following section on real-life examples, we successfully accomplish this task.

5.6.2 Benchmark Evaluation

In this section, we demonstrate the benefits or our LSQ on applications occurring in different domains which exhibit irregular and conditional data-dependences:

(1) *Histogram* calculates the histogram of an array of features; we discussed this example in Chapters 1 and 3. In each loop iteration, the value of one of the histogram bins needs to be increased, based on the input data and its corresponding weight. The loop may contain an inter-iteration read-after-write dependence if one of the following iterations needs to read the same histogram bin that a previous iteration is writing into (similar to the example discussed in Figure 5.1). Although the inter-iteration dependence rarely occurs, a static HLS tool cannot determine when they are present. Hence, it creates a conservative schedule with an initiation

interval equal to the number of cycles needed to write into the histogram bin of one iteration before reading a bin in the next iteration.

(2) *Maximal matching* is a graph algorithm which iterates through the edges of a graph and checks them for matching, i.e., determines if two edges share a common vertex. In case they are determined independent, the algorithm computes the new vertex values and updates the vertices using conditional stores. The kernel contains conditional loop-carried dependences which exist only when the stores need to be executed. In such cases, the HLS tool conservatively stalls the beginning of the next loop iteration until the stores have been completed.

(3) *Matrix power* multiplies the elements of a sparse matrix and a vector. In each iteration of a nested loop, a row and a column coordinate are read from memory and the corresponding matrix element is updated. As inter-loop read-after-write dependences can occur, the HLS tool fully sequentializes every loop iteration. We discussed this example in Chapter 3.

We compare our dataflow circuits (generated and optimized as described in the previous chapters) with an LSQ with a statically scheduled design of the same computation, which we generate using Vivado HLS. To provide an accurate and fair comparison of our designs against the one generated by Vivado, we employ the same arithmetic units produced and used by the HLS tool into our dynamic designs. The LSQ is connected to the very same memory subsystem that Vivado creates, thus limiting overall differences to a minimum. We form the groups for our designs corresponding to the basic blocks of the application. In the three explored designs, this strategy results in the following: (1) the *Histogram* application has a single group connected to the LSQ containing one load and one store port, (2) *Maximal matching* has two groups connected to the LSQ; one of the groups contains two load ports, and the other contains two store ports, (3) *Matrix power* contains two load ports and one store port in a group.

Exactly as Vivado HLS does and as explained at the end of Section 5.3, we connect the ports to different memories or different LSQs if it can be proven that they cannot alias; we will discuss our methodology to automatically determine this property in Chapter 6. This is the case for the weight values in *Histogram*, for the edges in *Maximal matching*, and for the row and column indexes in *Matrix power*.

Figure 5.14 shows the timing and resource utilization of the reference HLS designs together with our dynamically scheduled designs connected to LSQs of different sizes. We calculate the total execution time as the product of the clock period, obtained from the Vivado post-routing analysis, and the number of cycles, acquired by simulating the designs in Modelsim.

In all applications, the HLS tool creates the worst-case schedule with a conservative II. In contrast, our LSQ dynamically resolves memory access dependences, improving processing efficiency while increasing throughput and lowering execution time. Even with the increased cycle time, particularly for some LSQ sizes, the difference in cycle time is sufficiently small compared to the potential improvement in II. With a still affordable resource cost (i.e., the design with an LSQ of depth 8 occupies only under 5% of a typical FPGA), one can attain almost the

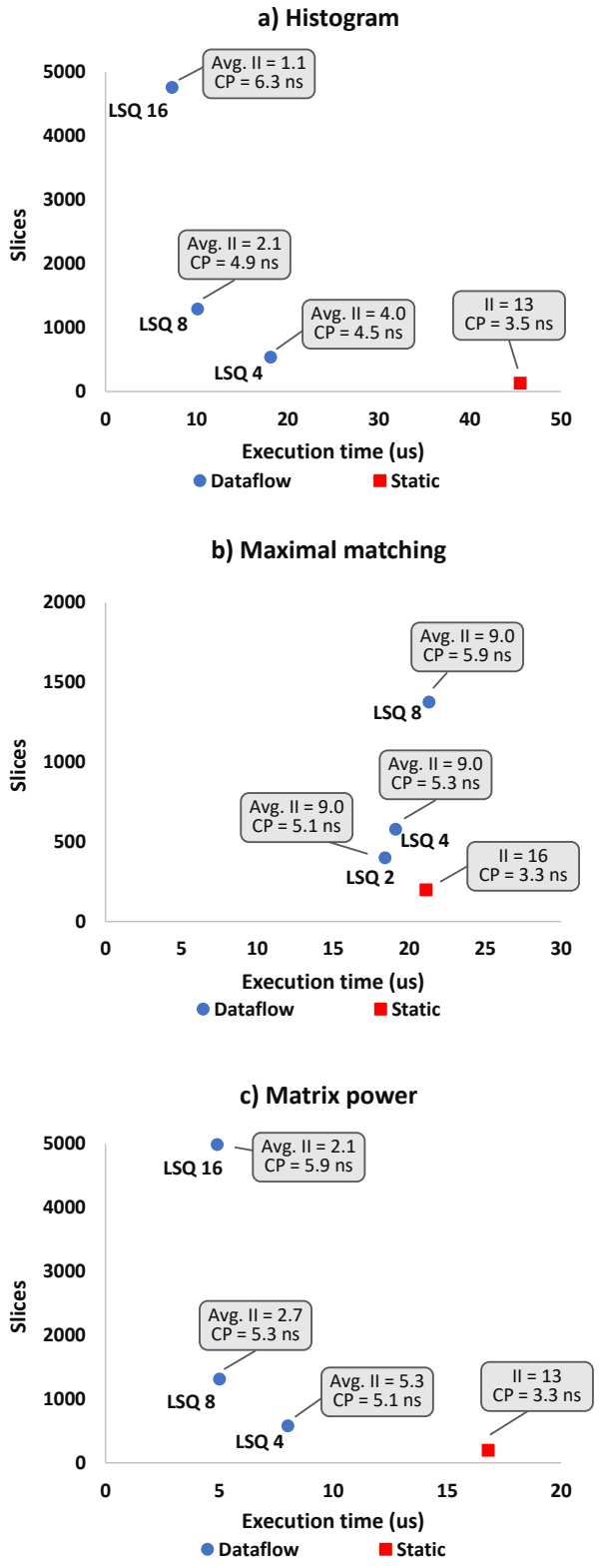


Figure 5.14 – Execution time and resource utilization of the static designs compared to the dataflow designs with the LSQ in different sizes (indicated in the labels next to the points). Although the LSQ designs increase the resource requirements, they achieve significant speed-ups over the static designs.

maximum parallelism available in the application. It is perhaps worth noticing that all discussed applications consist of little more than the memory accesses and thus require very few FPGA slices, making our LSQ large in comparison but not necessarily in absolute terms.

5.7 Conclusions

Dataflow execution naturally calls for memory interfaces capable of dynamically reordering accesses based on actual dependences through memory. Traditional LSQs for out-of-order processors seem the right component for such memory interfaces; yet, dataflow circuits lack the notion of program order which is necessary for their correct behavior. In this chapter, we discussed possible LSQ allocation strategies and focused on a novel policy which we believe is perfectly matched to the information available in HLS tools and silicon compilers. We presented a microarchitecture for such an LSQ and showed its ability to efficiently handle unpredictable memory accesses. Such memory interfaces are crucial for dataflow circuits to fully benefit from out-of-order execution.

6 Minimizing the Use of LSQs in Dataflow Designs

In the previous chapter, we demonstrated how to build a load-store queue which dynamically resolves memory dependences and enables dataflow circuits to correctly handle out-of-order memory accesses. While issuing every load and store request to memory through an LSQ guarantees correctness, this solution is unattractive: LSQs incur significant resource overheads as well as power and clock degradation with queue size; we have observed these effects in Section 5.6. Although standard HLS optimizations can, in certain cases, help in reducing this cost, analyzing memory access patterns of a dataflow circuit to decide where an LSQ is redundant remains a challenge—there is no predetermined schedule to provide information on the temporal ordering of memory accesses, so any two accesses targeting the same memory location may potentially conflict and, therefore, may require an LSQ. In this chapter, we explore a novel memory analysis technique which, based on the flow of data through the circuit, determines the activation ordering of certain memory accesses. This information allows us to rule out specific dependences and simplify the memory interface accordingly, leading to significant area savings and a tangible timing advantage.

6.1 Motivation

The code in Figure 6.1 shows a loop with multiple memory accesses which are analyzed and optimized using different memory analysis techniques; we already introduced this example in Section 2.4.4. The information about the memory accesses available in each analysis step is illustrated on the top of the figure: the green dashed edges indicate a possible dependence among accesses which the memory interface must appropriately handle (i.e., if two accesses are certainly or possibly dependent, they require an LSQ). The memory interfaces obtained in different analysis steps are illustrated below.

Without any memory analysis (Figure 6.1a) to reason about actual memory dependences, all accesses must connect to a single, large LSQ. This solution is correct, yet it is extremely resource-

This chapter is based on the work published at the *IEEE International Conference on Field Programmable Technology*, 2019 [69].

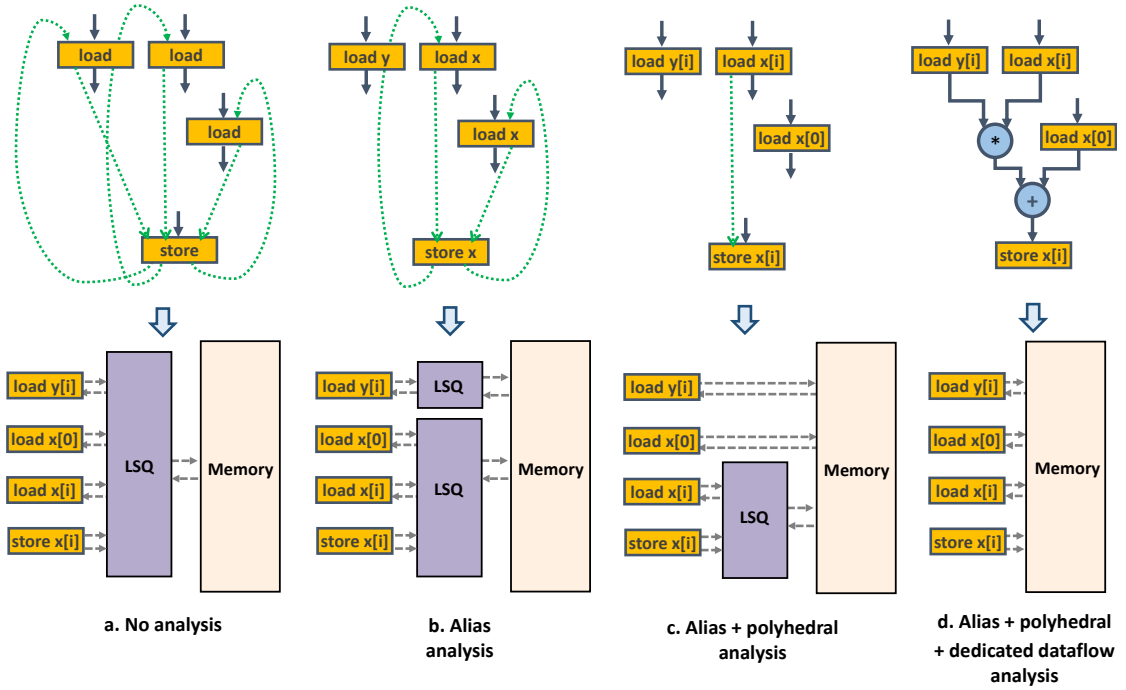


Figure 6.1 – Memory interface configurations of a dataflow circuit. Potential memory dependences between the memory accesses are indicated with green dashed edges. The circuit can be naively connected to memory using a single, large LSQ (Figure 6.1a). Standard memory optimizations allow us to disambiguate memory accesses targeting different memories (Figure 6.1b) as well as those which provably do not collide with any other access (Figure 6.1c). The optimal configuration (Figure 6.1d) is obtained using our specialized analysis for dataflow circuits. In this case, our analysis concludes that, since the load *must* occur before the store to the same memory location (i.e., the load is a certain producer of data for the store), the two accesses naturally occur in order and the LSQ can be omitted.

expensive, as it requires a large queue to maintain high parallelism and to consume all incoming requests at a high rate.

By exploiting methods such as alias analysis, one can disambiguate memory accesses which target different memories or memory regions and connect these accesses to independent memory ports using multiple, smaller LSQs (Figure 6.1b). Analyzing memory access patterns using techniques such as polyhedral analysis would allow us to simplify the design even further by removing LSQs in cases where the loads and stores targeting the same memory provably never access the same memory locations (Figure 6.1c). However, whenever this is not the case (i.e., the loads and stores might access the same locations at some point in time), standard techniques do not allow us to optimize our design any further—in a dataflow circuit, accesses may arrive at the memory interface out of order and an LSQ is needed to prevent a hazard. The main contribution of this chapter is an analysis which particularly targets dataflow circuits and, based on the flow of data through the dataflow graph, enables us to rule out specific data hazards through memory. In our example, it proves that a violation of the write-after-read dependence is impossible—the

load of `x[i]` is the data producer for the store of `x[i]` and the two accesses can therefore never occur out of order. This information enables us to completely remove the LSQ, as shown in Figure 6.1d.

In the rest of this chapter, we discuss our methodology for optimizing the memory interface of dataflow circuits. In Section 6.2, we discuss existing memory optimization techniques which more conventional forms of HLS regularly exploit. In Section 6.3, we introduce our new technique for analyzing memory accesses in an out-of-order dataflow circuit. We evaluate the effectiveness of our technique to simplify the memory interface in Section 6.4.

6.2 Background

This section provides an overview of standard memory analysis techniques which we will use.

6.2.1 Alias Analysis

The memory interface illustrated in Figure 6.1a corresponds to that of a compiler which does not perform any analysis of the memory accesses. Such a compiler must assume that each access in the code can point to any addressable value in memory [110] and therefore conservatively connects all accesses of the circuit to memory using a single monolithic LSQ.

Alias analysis groups pointers into sets such that different groups never access the same memory locations [110, 4]. HLS tools typically rely on alias analysis to simplify the memory interface by connecting different alias groups to different memories or independent memory ports. In our case, alias groups would connect to independent LSQs (see Figure 6.1b) which either insist on a single memory system (e.g., in FPGA cloud applications) or on different memories (e.g., separate block RAMs, like standard HLS tools employ [119]). Without loss of generality, we indicate in Figure 6.1 a single monolithic memory system with appropriate arbitration between ports.

6.2.2 Polyhedral Analysis

The polyhedral model is a linear-algebraic representation of a program which provides analysis capabilities for *Static Control Parts* (SCoPs). SCoPs are side-effect free regions of a program in which all control flow decisions and memory accesses are known at compile time [57].

The loop from Figure 6.1 is a SCoP in which the iterator i is bound by the constraints $1 \leq i < n$ and increased by 1 in each iteration. Polyhedral analysis can generate the sequence of array indices (i.e., memory addresses) that each memory instruction will access, which gives us information on all *read-after-write* (RAW), *write-after-write* (WAW), and *write-after-read* (WAR) dependences. In the example in Figure 6.1, there is a WAR dependence between the load and the store access of the same iteration. Because the load of `x[0]` only accesses the disjoint set $\{0\}$, we can simplify the memory interface to achieve the configuration from Figure 6.1c.

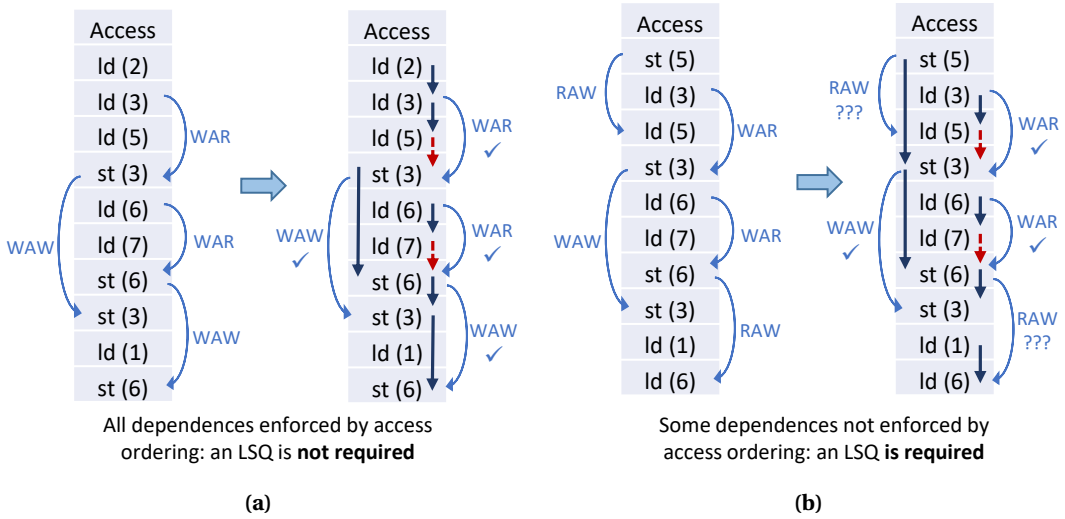


Figure 6.2 – Memory traces of two programs with a single load and a single store instruction. The programs contain RAW, WAR, and WAW dependences; the figures on the right show how we can exploit the ordering information between accesses to conclude that certain dependences will always be honored. Our analysis identifies data dependences between instructions (dashed arrows) which exclude access reordering and eliminate the need to use an LSQ.

6.3 Memory Interface Optimizations

It should be clear from the previous section that standard analysis techniques allow us to group memory instructions into a maximal number of *memory sets* such that the address of an instruction in one given set can never collide with the address of an instruction in another set (i.e., an LSQ is not needed across sets). Yet, without further analysis, each memory set with more than a single instruction *must* employ an LSQ to handle all RAW, WAW, and WAR dependences between the instructions; this is the very situation shown in Figure 6.1c. In the rest of this section, we will describe our original effort to simplify the memory interfaces of dataflow circuits past this design point, as suggested by Figure 6.1d.

6.3.1 The Ordering Problem

Let us assume that, using standard analysis techniques, we have clustered all memory instructions into a maximal number of mutually independent memory sets, as indicated above. We will focus here on a program with only two instructions in a single set; we will generalize our approach to sets with multiple instructions in Section 6.3.4. The same reasoning we here describe applies to programs with multiple sets by considering each set independently from the others.

We consider a program with a single load and a single store instruction in a memory set. In fact, the ordering relations between load and store instructions will be the main focus of our analysis; although the ordering of colliding memory accesses of two store instructions matters as well, our analysis will not achieve any memory interface simplification in this case, as we will

later observe. On the other hand, the ordering of a pair of load instructions does not impact correctness as none of the load accesses modifies the memory state.

Figures 6.2a and 6.2b show examples of sequential memory traces of two independent programs, which both contain a load and a store instruction. The purpose of a dynamically scheduled dataflow circuit is to execute each instruction as soon as its arguments are known, exactly as in a superscalar out-of-order processor [64]. Therefore, we generally need to assume that the circuit might try to reorder the shown sequences in any possible way. Some of these reorderings might result in semantically incorrect execution, as indicated in the figure. For instance, both sequences feature WAR dependences (i.e., there is a store access which writes into the same memory address that some load which precedes it in program order reads) and WAW dependences (i.e., there is a store access which writes into the same memory address as some store which precedes it in program order). The second sequence also contains RAW dependences (i.e., there is a load access which reads from the same memory address into which some preceding store writes). Although all other accesses can be reordered in the interest of execution speed, dynamically scheduled processors [64] and circuits need an LSQ to enforce correct ordering across the accesses with dependences. If we could otherwise ensure the correct ordering of these accesses, we would be able to omit the LSQ.

6.3.2 Exploiting Data Dependences

To remove the LSQ from the memory interface of a pair of load-store instructions, we need to reason about the ordering of their accesses in the execution of a dataflow circuit. There are two sources of information on which we can rely.

Firstly, each instruction performs its accesses strictly in program order—this property is guaranteed by the construction strategy of the dataflow circuit, as detailed in Section 2.3.3. Secondly, there might be a data dependence between a load access and the following store access which would guarantee their correct ordering—if the load produces the data necessary to compute the store value, there is no way for the store to run ahead in program execution.

The first property directly guarantees that WAW dependences are never a concern among write accesses of a single store instruction. The combination of the first and the second property ensures that any WAR dependence of a store access with *any* prior load access is maintained: if the store access has a data dependence with the preceding load access, and the load accesses execute in order, all previous load accesses will have completed before the store access, thus honoring any WAR.

However, the properties above do not help us reason about RAWs: there is no data dependence to rely on because the store is not a data producer for any other instruction, including the load. A store access may be arbitrarily ordered with respect to some subsequent load access and an LSQ is required to ensure correctness if their addresses collide.

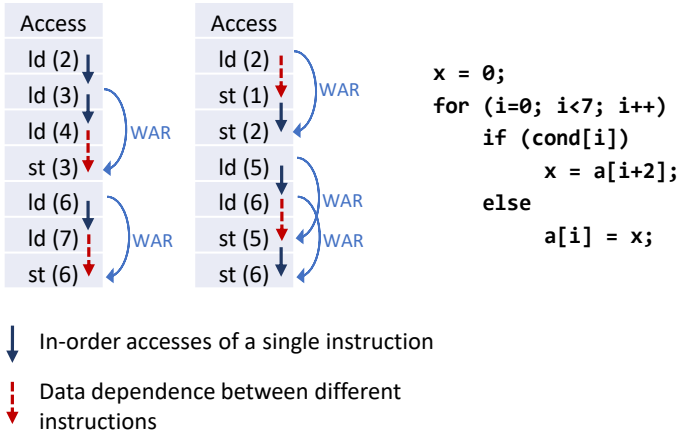


Figure 6.3 – Two (out of many) possible memory traces of the code in the figure. Because a data dependence will *always* occur between a load access and its subsequent store access, the circuit does not require an LSQ.

In summary, a pair of load-store instructions does not require an LSQ among themselves if (1) all WAR dependences of the original program are enforced by a data dependence between each store access and its preceding load access in the sequence and (2) there are no RAW dependences between the load and the store instruction. As determining the presence and absence of RAW dependences, among others, is often possible using standard analysis (as discussed in Section 6.2), we will here focus on the first property which is original for this work.

The right sequences of Figures 6.2a and 6.2b illustrate how the two properties above enable us to exclude certain dependences, assuming that a data dependence exists. In Figure 6.2a, both properties hold and guarantee that all dependent accesses are correctly ordered, so an LSQ is not needed. However, in Figure 6.2b, the second property does not hold and an LSQ is required.

Both conditions for omitting the LSQ hold for the example from Figure 6.1: each iteration has a WAR dependence between the load and the store access of the same iteration which is always honored because the load produces the data for the store. Furthermore, there are no RAW dependences in the program. Of course, this is a trivial case where the load and the store instruction are directly connected through the very datapath of the loop. A more interesting example is given in Figure 6.3. This code exhibits many possible memory traces (depending on the if-condition) which contain WARs across different (and not necessarily consecutive) loop iterations. To make sure that WARs are honored, we need to make sure that *all* possible traces have a data dependence between any load access and the store access that immediately follows. We will formalize this property in the following section.

6.3.3 Global Instruction Dependence

In this section, we describe the property of two instructions which guarantees that their order in the dataflow execution is equivalent to their order in the sequential program execution. As

described in Chapter 2, we consider a CFG of a program which is organized into BBs such that each BB contains a DFG of instructions (we denote the BB that instruction I belongs to as BB_I).

The dependence property that we will introduce applies to the *induced DFG* of a walk through the program CFG, where, as customarily, a *walk* is any sequence of BBs directly connected by control edges in the CFG (contrary to a path, a walk admits to visit every BB and traverse any edge an arbitrary number of times).

Definition 1. An *induced DFG* of a CFG walk w, denoted as $DFG_{ind}(w)$, is a DFG composed of a succession of the DFGs of the BBs traversed in w, repeated as many times as each BB is visited; the live-ins of each DFG are connected exclusively to the live-outs of the predecessor DFG.

Our analysis aims to determine *Global Instruction Dependence* (GID) of a pair of instructions.

Definition 2. Instructions N and M are *globally dependent* (written as $N \xrightarrow{\text{GID}} M$) if, for every CFG walk w starting with BB_N, ending with BB_M, and containing BB_N and BB_M only once, N is the predecessor of M in $DFG_{ind}(w)$.

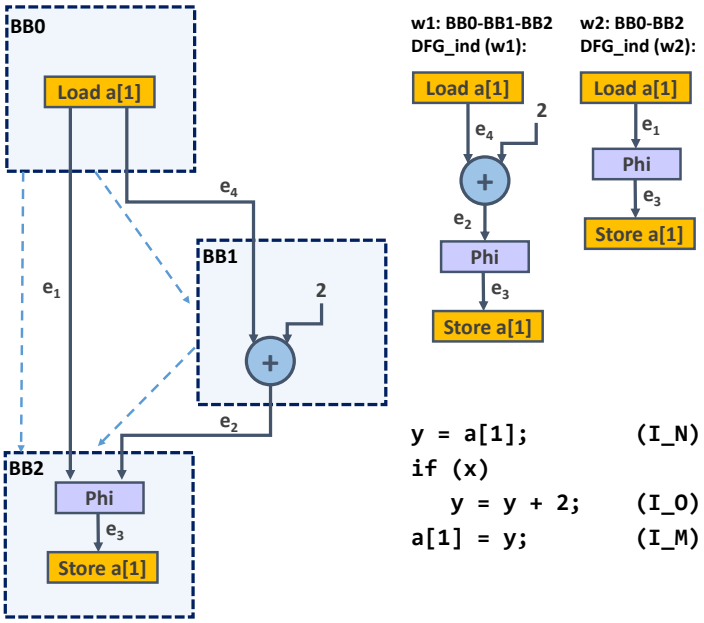
This property implies that, for every possible control flow sequence in which N and M execute, there is always a data dependence between N and M which enforces their in-order execution.

To illustrate, consider the examples in Figure 6.4. The memory instructions I_N and I_M belong to BB_0 and BB_2, respectively. The CFG objects and edges are shown in dashed in the figure. To determine whether a dependence relation exists between I_N and I_M, we need to consider all walks from BB_0 to BB_2, which are equivalent for both examples in the figure: $w_1 = [BB_0, BB_1, BB_2]$ and $w_2 = [BB_0, BB_2]$. The DFG_{ind} of each walk is shown in the figures. In both $DFG_{ind}(w_1)$ and $DFG_{ind}(w_2)$ of Figure 6.4a, I_N is the predecessor of I_M (i.e., the value to be written by I_M has a data dependence on the load I_N). Regardless of which CFG path is taken, the execution of I_M implies that I_N must have executed already, so $I_N \xrightarrow{\text{GID}} I_M$ and the WAR dependence between the two instructions is always honored. The same relation does not hold for the same instructions in Figure 6.4b, because I_N is not the predecessor of I_M in $DFG_{ind}(w_1)$. If walk w_1 is executed, there is no way to guarantee the ordering of these two instructions—the store may execute before the load which would result in a data hazard.

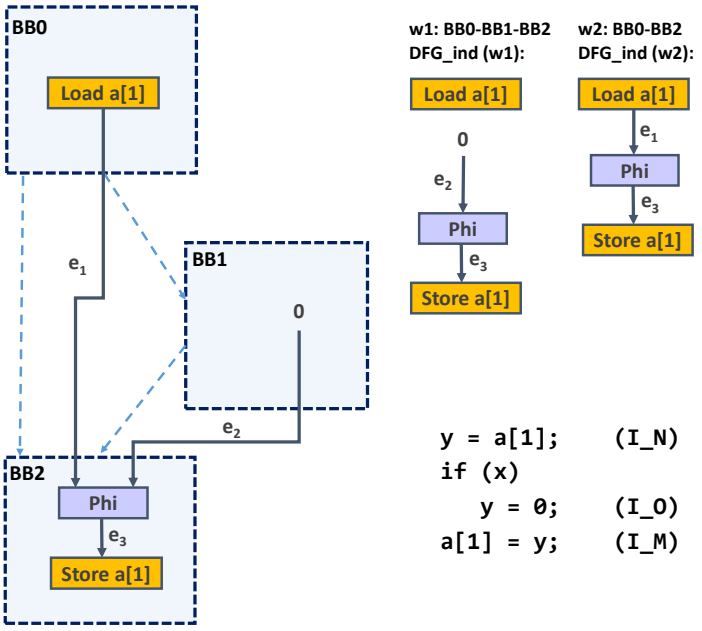
To summarize, instruction dependence ($L \xrightarrow{\text{GID}} S$) ensures that any WAR dependence between a load instruction L and a store instruction S is honored for every possible execution of the program. As mentioned earlier, we cannot exploit the same property to reason about RAW and WAW dependences, i.e., it will never hold that $S \xrightarrow{\text{GID}} L$ or $S \xrightarrow{\text{GID}} S$ because a store instruction never produces data and cannot be a predecessor of any instruction.

6.3.4 From Two Memory Instructions to Many

The properties from Section 6.3.2 with the original one formalized in Section 6.3.3 provide us with the knowledge about the activation order of a certain load-store pair and enable us to guarantee correct execution between them. Our algorithm, illustrated in Algorithm 6.1, exploits



(a)



(b)

Figure 6.4 – Code snippets and their control/data flow graphs which we use to illustrate the global instruction dependence property in Section 6.3.3. In the circuit in Figure 6.4a, the dependence property $I_N \xrightarrow{\text{GID}} I_M$ holds. This is not the case for the two instructions in Figure 6.4b because the load is not a predecessor of the store along both control flow paths.

```
// Input:  M (set of memory accesses with RAWs, WARs, and WAWs)
// Output:  M (minimized memory access set which requires an LSQ)

// Remove loads from set
foreach load L ∈ M do
    // remove:  indicates whether L should be removed from M
    remove = true
    foreach store S ∈ M do
        // If any of the WARs is not enforced by GID, L remains in M
        if WAR (L,S) and not L --GID--> S then
            remove = false

        // If L has a RAW with any store, L remains in M
        if RAW(L,S) then
            remove = false

    // Removing L from M
    if remove then
        RemoveFromSet (L, M)

// Remove stores from set
foreach store S ∈ M do
    // remove:  indicates whether S should be removed from M
    remove = true
    foreach load L ∈ M do
        // If S has a RAW or WAR with any load, S remains in M
        if WAR (L,S) or RAW (L,S) then
            remove = false
    foreach store S' ∈ M do
        // If S has a WAW with any store, S remains in M
        if WAW (S,S') then
            remove = false

    // Removing S from M
    if remove then
        RemoveFromSet (S, M)
```

Algorithm 6.1: Memory optimization based on global instruction dependence.

this information to reduce the number of instructions (i.e., the number of connections to an LSQ) of memory sets with more than two instructions as well.

Concretely, a load instruction L can be removed from memory set M if the two properties introduced in Section 6.3.2 hold for each store instruction S of M:

1. $L \xrightarrow{\text{GID}} S$, i.e., any WAR dependence between L and S is enforced by a data dependence.

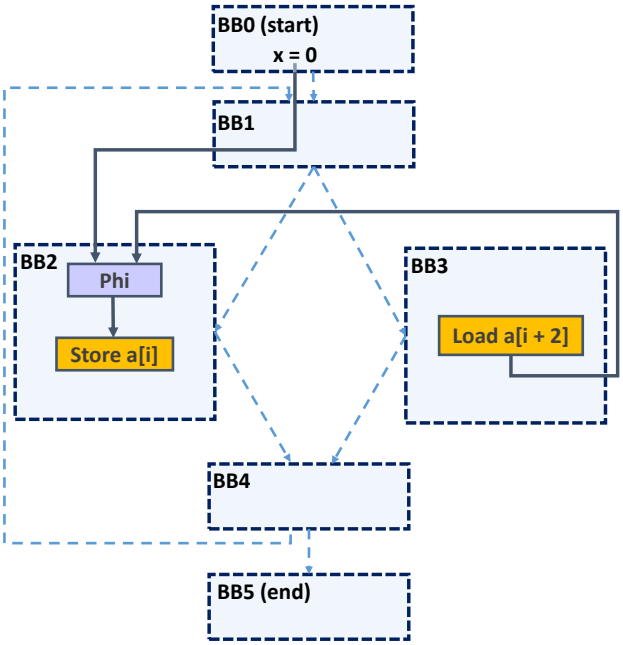


Figure 6.5 – Control/data flow graph of the example in Figure 6.3.

2. There are no RAW dependences between any of the accesses of L with any of the accesses of S.

If all WAR dependences between the load L and every store of the memory set are provably maintained in order and there are no RAW dependences with any store of the set, L does not need an LSQ and can be removed from the set.

After certain load instructions have been removed from a memory set using the properties above, it is trivial to re-evaluate the set to remove all store instructions which no longer have conflicting accesses with any of the remaining instructions in the set.

6.3.5 Why Not CFG Dominance?

If one is familiar with classic compiler analysis, it may appear that our Global Instruction Dependence should, in fact, be the classic notion of CFG dominance and post-dominance. These notions describe the relations between BBs in a CFG as follows:

1. A basic block BB_N *dominates* basic block BB_M if every path from the entry of the graph to BB_M must go through BB_N.
2. A basic block BB_M *post-dominates* basic block BB_N if all paths to the exit of the graph starting at BB_N must go through BB_M [110].

It might seem that we could use these properties to describe a relationship between a load and a store instruction. However, they would not suffice, as the examples in Figure 6.4 clearly illustrate:

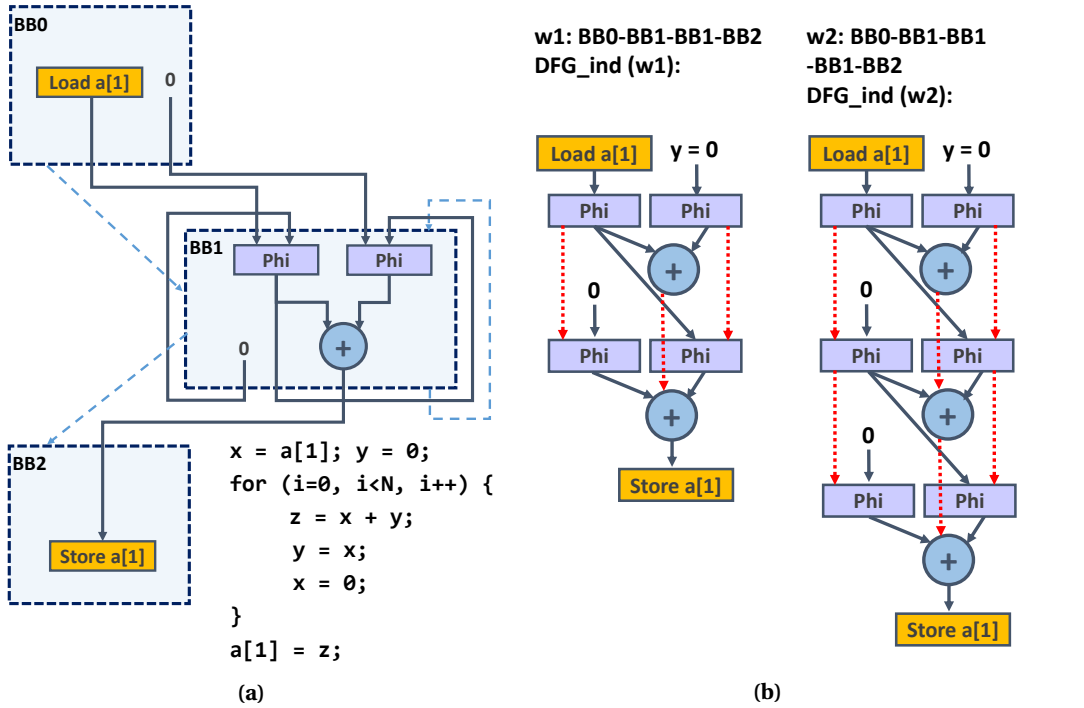


Figure 6.6 – Another ordering guarantee. The circuit in Figure 6.6a exhibits an absence of load-store dependence only after two traversals of the CFG cycle through BB_1, as illustrated in Figure 6.6b (path w_2). However, multiple instances of the same instruction are guaranteed by construction to execute in order. These ordering dependences are indicated with dotted arrows; they provide additional dependences between instructions (in this case, a dependence between the load and the store in path w_2).

the CFGs of the two programs are the same (i.e., in both cases, BB_0 dominates BB_1 and BB_2, BB_2 post-dominates BB_0 and BB_1), yet the memory instructions exhibit different dependence relations, as discussed in Section 6.3.3. Therefore, determining instruction dependence using these CFG properties might lead to incorrect results.

One could consider formulating properties similar to dominance at the instruction level: informally, one could check whether, on each path from program entry to a store, one passes through the load, or whether every path from the load to the exit passes through the store. While not incorrect, this formulation would be restrictive. Consider the example from Figures 6.3 and 6.5: in this case, neither of these properties would hold (e.g., there is a path from the start of the program to the store which does *not* pass through the load); one would conclude that dominance does not exist and would conservatively place an LSQ. Our formulation is more general and captures situations when there is effectively a load before the store.

6.3.6 Another Ordering Guarantee

Our definition is quantified on all walks through the CFG which have a single passage through the BB of the load and the store. This constraint provides us with the ordering of the *last* execution of the load before the execution of the store—if the dependence relation holds for these two

accesses, all previous load accesses and all successive store accesses can be ordered with respect to this load-store pair. Yet, if the CFG has a cycle which does *not* contain the load or the store BB, the absence of a load-store dependence may be detectable only after multiple traversals of the cycle, as illustrated in Figure 6.6—the lack of data dependence is present only after two cycle traversals (i.e., in walk w_2) and, based on Section 6.3.4, would require an LSQ between the load and the store. But there is more.

As discussed in Section 2.3.3, accesses of the same instruction *always* execute in order—we have already used this property to order load and store accesses, but it applies to other instructions as well. This particularity adds implicit dependence edges between multiple instances (corresponding to multiple accesses) of the same instruction in induced DFGs and provides us with additional dependence relations.

Property. Each instruction instance in a $DFG_{ind}(w)$ has an ordering dependence on any preceding instance of the same instruction.

In Figure 6.6a, this property holds for the *phi* and *add* instructions which repeat (we indicate all ordering dependences with red dotted edges in Figure 6.6b); the ordering between the instances of the leftmost *phi* implies a dependence between the load and the store along *every* walk—in w_2, the load is not a data predecessor of the store, but the new ordering edges enforce their ordering and allow us to disconnect them from the LSQ. In the following section, we will formalize how this property enables us to assess dependences and to further simplify the memory interfaces.

6.3.7 How Long a Walk Does One Need?

Our current definition requires checking *all* CFG walks to determine load-store dependences; in certain cyclic CFGs, it requires checking an infinite number of walks, which is not quite feasible. In the rest of this section, we discuss how to use the property described in the previous section to restrict our search to a finite subset of walks, making the test practical.

Theorem. If instructions N and M are dependent on the $DFG_{ind}(p)$ of a path p from BB_N to BB_M, they are also dependent on $DFG_{ind}(w)$ of any walk w from BB_N to BB_M which contains p (i.e., which includes all BBs and CFG edges belonging to path p).

We here consider a path to be a walk in which all vertices are distinct, except possibly the first and the last [15].

Proof. Given a path p from BB_N to BB_M through any BB_P, $p = [BB_N, ..., BB_P, ..., BB_M]$, a dependence between N and M (denoted as $N \rightarrow M$) on $DFG_{ind}(p)$ implies that there exists at least one instruction P in BB_P such that $N \rightarrow P$ and $P \rightarrow M$. Let us consider now a walk w which covers a single CFG cycle and contains p; the cycle has a CFG edge which connects into p in BB_P; w visits BB_P multiple times, that is, $w = [BB_N, ..., BB_{P_{\text{first}}}, ..., BB_{P_{\text{last}}}, ..., BB_M]$, where $BB_{P_{\text{first}}}$ is the first visit to BB_P in the walk and $BB_{P_{\text{last}}}$ is the last one. The dependence over the path p ensures that $N \rightarrow P_{\text{first}}$ and $P_{\text{last}} \rightarrow M$. The property of Section 6.3.6 also guarantees $P_{\text{first}} \rightarrow P_{\text{last}}$. Thus, transitively, $N \rightarrow M$ also on $DFG_{ind}(w)$. The same reasoning applies for all walks with

whatever cycles or control flow merges into p: a dependence holds across each BB where w connects into p and transitively holds across all of them. ■

This theorem restricts Definition 2 in Section 6.3.3 to paths instead of generic walks—we hence refine it to formulate *Global Instruction In-order Dependence* (GIID):

Definition 3. Instructions N and M are *globally in-order dependent* (written as $N \xrightarrow{\text{GIID}} M$) if, for every CFG path p starting with BB_N, ending with BB_M, and containing BB_N and BB_M only once, N is the predecessor of M in $DFG_{ind}(p)$.

The decision to remove a load instruction L from a memory set can accordingly account for GIID instead of GID (see the first property of Section 6.3.4): if $L \xrightarrow{\text{GIID}} S$, any WAR dependence between L and S is enforced by a data dependence. We rely on this formulation in our experiments in Section 6.4.

6.4 Evaluation

In this section, we demonstrate the ability of our memory analysis to reduce the area and performance cost of the memory interface of dataflow circuits.

6.4.1 Memory Analysis Implementation

We implement our memory optimization, detailed in Section 6.3, as an LLVM pass which determines the appropriate memory interface of a dataflow circuit. We exploit the alias analysis pass of LLVM (i.e., BasicAA [86]) to determine whether two pointers alias and to disambiguate arrays that target different memory regions. To extract the information about the memory access patterns, we rely on the ScopInfo pass of the Polly framework [58]. This pass detects SCoP regions within a program and creates a polyhedral description of the memory accesses it contains. We use this information to determine whether two instructions have RAW, WAW, and WAR dependences. Whenever an instruction is not part of any SCoP (i.e., Polly does not provide us with the memory access patterns), we connect it to an LSQ to ensure correctness.

To determine whether a dependence relationship holds between a load-store pair, we employ depth-first search to find all CFG paths from the load to the store BB. We extract the DFG_{ind} of each path and perform a traversal across it to determine whether the load precedes the store. The information on the memory access patterns and the GIID relation determined in this step (see Section 6.3.7) enable us to simplify the memory interfaces, as described in Section 6.3.4.

6.4.2 Experimental Methodology

In the rest of this section, we compare three design points: (1) designs without any optimization of the memory interface, which qualitatively correspond to the interface of Figure 6.1a, where

Table 6.1 – Memory access patterns of our benchmarks. The loop iterators are indicated as i and j. Function *func* is application-specific.

Benchmark	Memory access pattern
Memory loop	x[i] = *func* (x[0], x[i], y[i])
Scalar multiply	x[i] = *func* (x[i])
Image revert	x[i][j] = *func* (x[i][j])
Weighted sum	x[i] = *func* (x[i], x[i-1], x[i+1], y[i], y[i-1], y[i+1])
Threshold	{x[i], y[i], z[i]} = *func* (x[i], y[i], z[i])
Video filter	x[i] = *func* (x[i]), y[i] = *func* (y[i]), z[i] = *func* (z[i])
Histogram	x[y[i]] = *func* (x[y[i]], z[i])
Matrix power	x[i][y[j]] = *func* (x[i][y[j]], x[i-1][w[j]], z[i])

all memory accesses of the circuit are connected to memory using a single LSQ; (2) designs optimized using the information provided by standard techniques (i.e., alias analysis and polyhedral analysis), and (3) our designs which, in addition to standard information, use the methodology from Section 6.3 to simplify the memory interface.

Our circuits are synthesized automatically from C code using the strategies from Chapters 2 and 3; we use the LSQ implementation from Chapter 5. We manually choose the minimal power-of-2 size which allows for maximum achievable loop parallelism (i.e., all designs are pipelined to achieve the optimal loop initiation interval for the particular benchmark and memory configuration). We connect disambiguated memory instructions to separate dual-port BRAMs with a single-cycle read and write latency, either directly or through an LSQ.

We functionally verify all designs using ModelSim. We obtain the number of clock cycles from the simulation and the clock period (CP) from the post-routing timing analysis with Vivado to calculate the execution time. Vivado place-and-route gives us the resource usage (i.e., the FPGA slices, LUT, FF, and DSP count). All designs target a Xilinx Kintex-7 FPGA.

6.4.3 Benchmarks

The designs that we evaluate are simple but realistic kernels from literature [53, 80, 40] which contain different memory access patterns, summarized in Table 6.1: (1) *Memory loop* is the example from Figure 6.1 which we have discussed in Section 6.1. (2) *Scalar multiply* reads the values of a vector and rescales each value by a constant factor before storing it back into the same memory location. (3) *Image revert* is a nested loop in which the value of each image pixel (stored in a 2-dimensional array) is reverted by subtracting it from a constant. (4) *Weighted sum* updates each value of a vector to the weighted sum of itself and its neighboring vector values. The weights corresponding to each element are stored in a separate vector. (5) *Thresholding* is an edge detection kernel with a conditional statement inside a loop: if the pixel intensity is less than some fixed constant, it is replaced with a black pixel. (6) *Video filter* is a simple video

Table 6.2 – Memory optimization comparison, timing results. Timing of dataflow circuits which exploit our memory interface optimization (*This Work*), compared to circuits with naively built memory interfaces (*No opt.*) as well as memory interfaces created using standard optimizations (*Standard*). The LSQs employed by each design are listed under *Interface*, together with the LSQ depth (e.g., d8 indicates a depth 8) and the number of connected ports (e.g., p2 indicates that two memory accesses connect to the LSQ).

Benchmark	Optimization	Interface	CP (ns)	Exec. Time (μs)	Speedup
Memory loop	No opt.	LSQ (d8, p4)	6.7	13.6	–
	Standard	LSQ (d8, p2)	6.3	9.5	1.4×
	This Work	–	4.3	4.4	3.1×
Scalar multiply	No opt.	LSQ (d8, p2)	5.8	8.8	–
	Standard	LSQ (d8, p2)	5.8	8.8	1.0×
	This Work	–	4.0	4.0	2.2×
Image revert	No opt.	LSQ (d8, p2)	6.3	5.7	–
	Standard	LSQ (d8, p2)	6.3	5.7	1.0×
	This Work	–	6.3	5.7	1.0×
Weighted sum	No opt.	LSQ (d8, p7)	6.5	71.5	–
	Standard	LSQ (d4, p4)	5.4	48.6	1.5×
	This Work	LSQ (d2, p2)	4.0	36.0	2.0×
Threshold	No opt.	LSQ (d4, p6)	13.6	136.5	–
	Standard	3 LSQ (d2, p2)	11.1	88.9	1.5×
	This Work	–	11.1	33.4	4.1×
Video filter	No opt.	LSQ (d16, p6)	8.8	23.9	–
	Standard	3 LSQ (d8, p2)	7.6	10.4	2.3×
	This Work	–	6.6	6.2	3.9×
Histogram	No opt.	LSQ (d16, p4)	7.0	35.2	–
	Standard	LSQ (d16, p2)	6.3	7.3	4.8×
	This Work	LSQ (d16, p2)	6.3	7.3	4.8×
Matrix power	No opt.	LSQ (d16, p5)	8.0	18.4	–
	Standard	LSQ (d16, p3)	6.2	4.8	3.8×
	This Work	LSQ (d16, p3)	6.2	4.8	3.8×

processing application which, in a nested loop, applies a filtering function on each video pixel. (7) *Histogram* calculates the weighted histogram, which has a potential RAW dependence which cannot be determined at compile time. (8) *Matrix power* computes a series of matrix-vector multiplications in a nested loop. The row and column coordinates of the read and written elements are unknown at compile time. We explored the last two benchmarks in Chapter 5.

6.4.4 Results

Tables 6.2 and 6.3 summarize our comparison results. All cases where no memory optimization is applied (*Naive*) need a single, large LSQ. The number of LSQ ports corresponds to the total number of reads and writes within the kernel. The execution time suffers due to two effects: (1) the large number of LSQ entries degrades frequency (see Section 5.6) and (2) multiple ports simultaneously insist on the same LSQ and, consequently, the same dual-port BRAM, which causes memory port congestion and limits parallelism.

Table 6.3 – Memory optimization comparison, resource utilization. Resources of dataflow circuits which exploit our memory interface optimization (*This Work*), compared to circuits with naively built memory interfaces (*No opt.*) as well as memory interfaces created using standard optimizations (*Standard*). The LSQs employed by each design are listed under *Interface*, together with the LSQ depth (e.g., d8 indicates a depth 8) and the number of connected ports (e.g., p2 indicates that two memory accesses connect to the LSQ).

Benchmark	Optimization	Interface	Slices	LUTs	FFs	DSPs
Memory loop	No opt.	LSQ (d8, p4)	1390	4755	1751	3
	Standard	LSQ (d8, p2)	1213 (-13%)	3892 (-18%)	1475 (-16%)	3
	This Work	–	107 (-92%)	316 (-93%)	282 (-84%)	3
Scalar multiply	No opt.	LSQ (d8, p2	1092	3506	1512	3
	Standard	LSQ (d8, p2)	1092 (-0%)	3506 (-0%)	1512 (-0%)	3
	This Work	–	100 (-91%)	262 (-93%)	317 (-79%)	3
Image revert	No opt.	LSQ (d8, p2)	1013	3425	1455	0
	Standard	LSQ (d8, p2	1013 (-0%)	3425 (0%)	1455 (-0%)	0
	This Work	–	123 (-88%)	392 (-89%)	287 (-80%)	0
Weighted sum	No opt.	LSQ (d8, p7)	1640	5272	2312	9
	Standard	LSQ (d4, p4)	628 (-62%)	1776 (-66%)	1442 (-38%)	9
	This Work	LSQ (d2, p2)	313 (-81%)	772 (-85%)	1013 (-56%)	9
Threshold	No opt.	LSQ (d4, p6)	440	1356	831	0
	Standard	3 LSQ (d2, p2)	350 (-20%)	917 (-32%)	847 (+2%)	0
	This Work	–	183 (-58%)	587 (-57%)	425 (-49%)	0
Video filter	No opt.	LSQ (d16, p6)	4356	15546	3596	9
	Standard	3 LSQ (d8, p2)	3408 (-22%)	11282 (-27%)	4366 (+21%)	9
	This Work	–	385 (-91%)	1073 (-93%)	933 (-74%)	9
Histogram	No opt.	LSQ (d16, p4)	5206	17860	3484	2
	Standard	LSQ (d16, p2)	4762 (-9%)	3562 (-5%)	3125 (+2%)	2
	This Work	LSQ (d16, p2)	4762 (-9%)	3562 (-5%)	3125 (+2%)	2
Matrix power	No opt.	LSQ (d16, p5)	5345	18394	3549	7
	Standard	LSQ (d16, p3)	4989 (-7%)	16955 (-8%)	3744 (+5%)	7
	This Work	LSQ (d16, p3)	4989 (-7%)	16955 (-8%)	3744 (+5%)	7

Standard memory analysis techniques (*Standard*) disambiguate memory accesses targeting different arrays. All applications with accesses to multiple arrays benefit from the reduced port count and, in certain cases, reduced LSQ depth. Consider, for instance, *Video filter*: the naive implementation had a large LSQ with 6 ports which connected three arrays to memory (i.e., x, y, and z, as indicated in Table 6.1); this optimization step splits the single LSQ into three smaller LSQs (one for each of the arrays). Apart from disambiguating accesses of different memories, standard analysis determines read-only accesses which can be connected to memory separately from the LSQ. This is the case, for instance, for `x[0]` in *Memory loop* and for accesses to arrays y and z in *Histogram* and *Matrix power*. However, in all applications, all read and write instructions which access the same memory locations still require an LSQ.

In all applications apart from *Histogram* and *Matrix power*, our technique (*This Work*) finds timing relations between instructions which enable us to simplify or to completely remove the LSQ from the memory interface, resulting in significant area savings (although remarkable, note

that the area savings due to the removal of the complex LSQ circuitry are probably exaggerated by the simplicity of the kernels we consider). In *Weighted sum*, an LSQ is still required to handle the loop-carried RAW dependence between the load of `x[i+1]` and store to `x[i]` (see Table 6.1), yet even a queue of minimal depth sustains maximal throughput. *Histogram* and *Matrix power* always require an LSQ to handle memory dependences which cannot be determined at compile time (i.e., the kernels are not SCoPs and polyhedral analysis cannot extract the accessed indices). These examples are representative of cases where dynamic scheduling of dataflow circuits is superior to static HLS and an LSQ is essential—in any static approach, the loops cannot be pipelined due to the possible read-after-write dependences; in contrast, dataflow circuits exploit the LSQ to maximize parallelism, as we demonstrated in Chapter 5.

6.5 Conclusions

Reordering memory accesses in dataflow circuits implies the use of LSQs, qualitatively similar to those used in processors. The problem is that spatial dataflow circuits essentially need an LSQ port for every access, quickly making LSQs prohibitive in cost and terrible in performance, especially on FPGAs; the risk of negating any advantage of dynamic scheduling is extremely concrete. It seems intuitive that one should be able to use information from the source code to disambiguate accesses and partition monolithic LSQs into smaller ones as well as bypass some of the LSQs. In this chapter, we described how to build such a network of LSQs for arbitrary applications. Our optimization exploits the fact that data dependences in the original code imply sequential execution of some accesses in the corresponding dataflow circuit; this guaranteed sequencing may remove the need for some LSQs or ports thereof. On average, we demonstrated that a careful design can reduce by 65% the design area and improve performance by a factor of 3.1× compared to a naive approach.

7 Speculative Dataflow Circuits

In the realm of processors, VLIWs have most suffered from the inability to accommodate arbitrary forms of speculative execution: predicated execution (i.e., committing an instruction only if a specific condition is true) can be seen as a form of speculation when used to implement if-conversion (two branches of an *if-then-else* statement are both executed until the value of the condition is known); yet, even aggressive predication is not applicable to every performance-critical control decision. Analogously, statically scheduled circuits generated by standard HLS tools suffer from the inability to exploit broad classes of speculative execution.

On the other hand, part of the success of speculative execution in dynamically scheduled processors is the fact that a fairly limited set of universal techniques (i.e., register renaming, reordering buffers, and commit mechanisms) is sufficient to support the speculation of virtually any critical decision worth predicting. In this chapter, we explore whether similarly broad classes of speculation can be supported in dataflow circuits. We demonstrate that this is indeed possible, that it also needs a fairly small number of generic components and techniques, and that the advantage can be significant when waiting for a key execution decision is particularly time-consuming.

7.1 Why HLS Needs Speculative Behavior

Figure 7.1 revisits the example from Section 2.4.5, which illustrates the need to accommodate speculative behavior. A standard, nonspeculative HLS tool would not allow a new loop iteration to start until the condition to exit the loop has been checked—this condition is available only after performing almost the entire loop body, which largely prevents pipelining of the loop. In contrast, speculation would make possible a high-throughput pipeline which tentatively starts another loop iteration on every clock cycle and, later on, discards the speculatively computed values if the loop was supposed to terminate prior to their execution (in the case of Figure 7.1,

This chapter is based on the work published at the *27th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2019 [74].

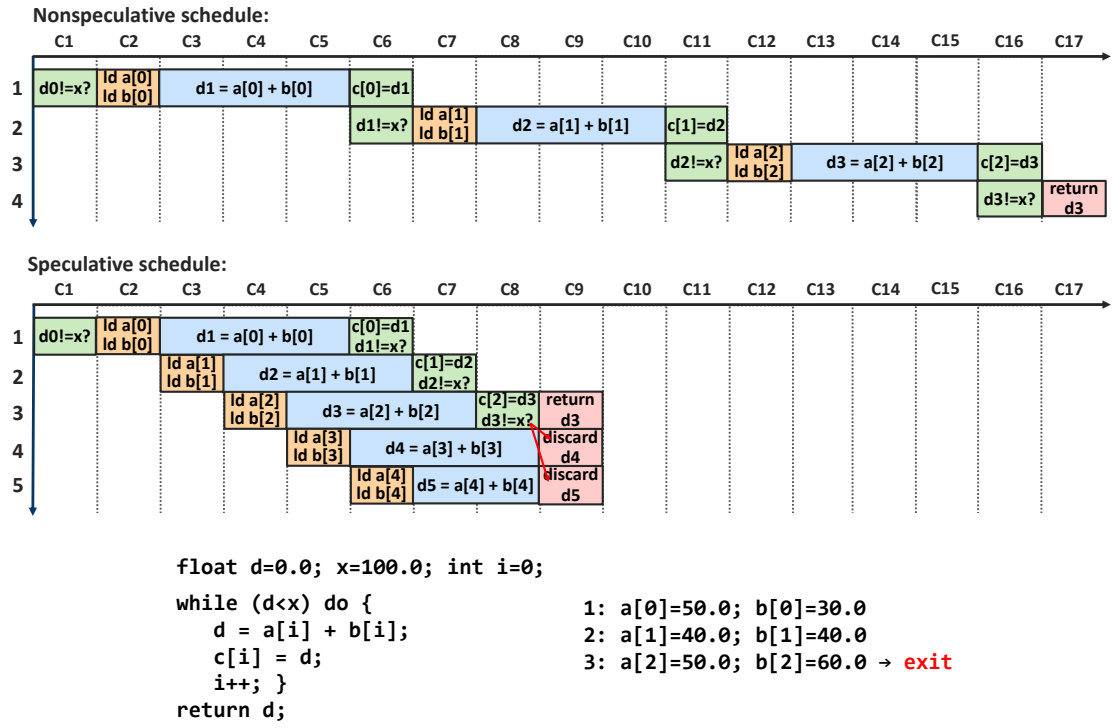


Figure 7.1 – A nonspeculative schedule, compared to a schedule produced by a system supporting speculative behavior. This figure repeats the situation from Figure 2.14. The code below the schedules takes multiple clock cycles to compute the condition for executing another loop iteration. A nonspeculative circuit needs to wait for the condition, whereas the speculative circuit tentatively starts another iteration and then discards the newly computed values if they are later on determined unneeded.

the addition results from the fourth and the fifth iteration are unneeded and will be discarded once this is decided by the termination check in cycle *C*7).

Figure 7.2 shows the dataflow circuit corresponding to the example in Figure 7.1; this simple program needs a single branch and a single merge node for the loop iterator `i`. The token carrying the iterator value is propagated to the next loop iteration whenever this is decided by the branch condition (i.e., the comparison of the value `d` with `x`). Despite the flexibility of dynamic scheduling, if the condition takes a long time to compute (as it is the case here), the branch unit will hold the token representing `i+1` until the token with the condition arrives, and the start of a new loop iteration will be delayed—speculative execution (in this case, branch prediction and speculation) is needed to achieve an efficient pipeline.

7.2 Speculation in Dataflow Circuits

Our goal is to create a generic framework for handling speculation in dataflow circuits. The idea is that some units might be allowed to issue *speculative tokens*—that is, pieces of data which might or might not prove correct and which will combine with other (nonspeculative) tokens, resulting in more speculative tokens traveling through the circuit. In other words, speculative

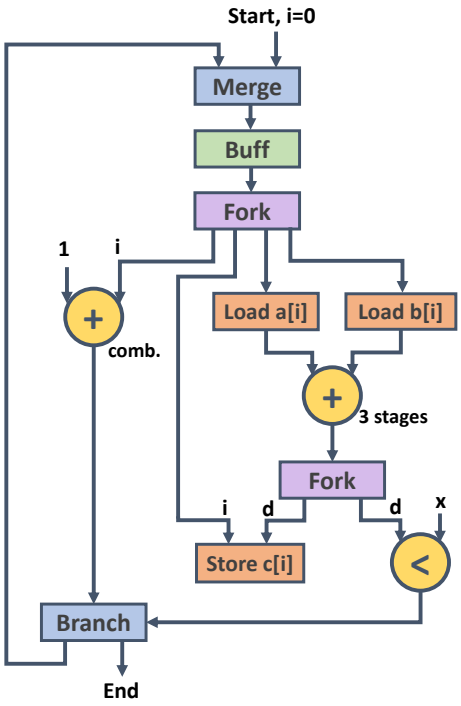


Figure 7.2 – A dataflow circuit executing the code of Figure 7.1. The branch requires both a value and a binary condition before it can issue the value it has received to the merge and thus start a new iteration. Hence, a condition which takes a long time to compute may significantly hinder performance.

tokens trigger some computations which might have to be squashed and possibly repeated with the correct nonspeculative tokens. Figure 7.3 gives a sense of our strategy: speculative tokens will be contained in a region of the circuit delimited by special units.

The first unit is a *speculator*. A speculator is a special version of a regular dataflow unit which, besides its standard functionality, also has the liberty of injecting tokens before receiving any at its input(s). The most natural example is that of a branch node which receives the value to dispatch but not the condition; a branch speculator could predict the missing condition and send tentatively the value through one of its outputs. If, after issuing a speculative token, the speculator eventually receives the same data which it assumed speculatively (e.g., the condition it predicted), all is fine and execution was probably sped up; if, on the other hand, the data it eventually receives does not match the prediction, we have a case of misspeculation: the speculator should now perform its function correctly (e.g., resend the value on the other output), but must first make sure that the speculative work done is discarded.

The reason for the output boundary of the speculative region of Figure 7.3 is fairly evident: clearly, speculative tokens cannot be allowed to propagate indefinitely and must not affect the architectural state of the circuit, i.e., the part of the state which is known and visible to the user. Therefore, the speculative region must be limited at least before units which store values in memory or before the end of the circuit. The units at the output end of the speculative region are called *commit* units. These units simply propagate further speculative results which turn

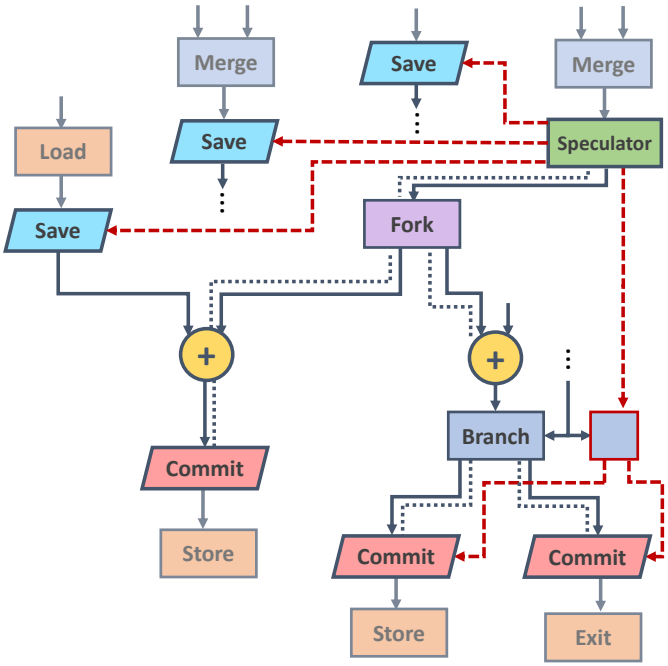


Figure 7.3 – A region of a dataflow circuit implementing our speculative execution paradigm. The speculator initiates speculative execution by injecting tokens tentatively, save units capture required inputs of the region to enable a correct replay in case of misspeculation, and commit units prevent speculative tokens from affecting irreversibly the architectural state, such as memory. Speculative tokens are marked explicitly using an additional bit (represented by the dotted line). A dataflow control circuit (in red, dashed line) between the speculator and the save and commit units carries information about speculative events (start, commit, squash, etc.).

out to be correct; as it happens in speculative software processors, misspeculated results are simply squashed. Because commit units must differentiate speculative from nonspeculative tokens (the former ones need explicit commit information before propagating, while the latter ones can always go ahead), as Figure 7.3 suggests, all channels between the speculator and the commit units must be enriched with a control signal which indicates the speculativeness of the token being passed.

Finally, we need to bound the speculative region on the input side in order to save a copy of all regular tokens which may combine with a speculative token so as to be able to reissue them if the previous computation is squashed. To this end, we employ *save* units.

Section 7.3 details these new dataflow units and Section 7.4 describes how to correctly place them in the circuit. An important aspect of a speculative region, i.e., the communication between the speculative units, is only sketched in Figure 7.3: the speculator should communicate with the commit and save units whenever it starts and stops a speculative event. We have elected to implement this communication through an additional dataflow circuit connecting all the new speculative units; while this communication is relatively straightforward (essentially, binary tokens indicating whether a speculation was successful or not), there are a few peculiarities to

take into account when speculative tokens traverse merge and branch units. We will detail the construction of this control circuitry in Section 7.5. A critical situation, not represented in the qualitative example of Figure 7.3, occurs when the speculator is placed on a loop: we will use an intuitive but too conservative approach in Section 7.4, likely to result in speculative circuits with little performance advantage, and then fully tackle this problem in Section 7.6.

7.3 Units for Speculation

This section details the units needed to delimit a speculative region in a dataflow circuit: a speculator to initiate the process, save units on the inputs of the region, and commit units on its outputs. The units are, in general, built out of standard dataflow units and they communicate with the rest of the design using the same handshake protocol.

7.3.1 Speculator

Speculative execution starts when a *speculator* triggers the execution of a part of the circuit before it is certain that it needs to execute or that the execution is correct. Any dataflow unit can operate as a speculator by issuing *speculative tokens* before all of the unit's input information is available (i.e., when only a subset of the input tokens is available at the inputs of the unit). For instance, a speculator branch can speculate on the condition, causing the branch to output the data token to one of its successors before the condition token arrives; a speculator within a load-store queue can perform a speculative load and eagerly output a speculative data token as soon as the load address is available and before all memory dependences are resolved.

Apart from issuing speculative tokens, the speculator's role is to determine the correctness of a speculation and trigger actions accordingly. It therefore saves the prediction and assesses the situation once the missing input arrives. It is important to note that the tokens propagate through each path of the circuit *strictly in order* (see Section 2.3.3); hence, the first token arriving at the particular input whose value was speculated will hold the value which resolves the first speculation. After deciding if the prediction was successful, the speculator informs the appropriate units in the circuit of the comparison result, allowing them to commit the speculative results or to discard the misspeculated tokens and recompute with the correct values. In the second case, the speculator needs to insert the token holding the correct value into the circuit in order for the computations to execute anew.

The structure of one of the most natural speculators, i.e. the branch speculator, is shown in Figure 7.4. Unlike a standard branch, which waits for both the data and the condition to arrive before producing an output, the branch speculator can output a data token even when the condition is not yet present, together with a bit indicating whether the token is speculative. Eventually, it will receive a regular condition token, which it will compare with the previously speculated value (all speculatively issued values are stored in a queue within the speculator) and send a confirmation or cancellation token to the other units. It will then either discard

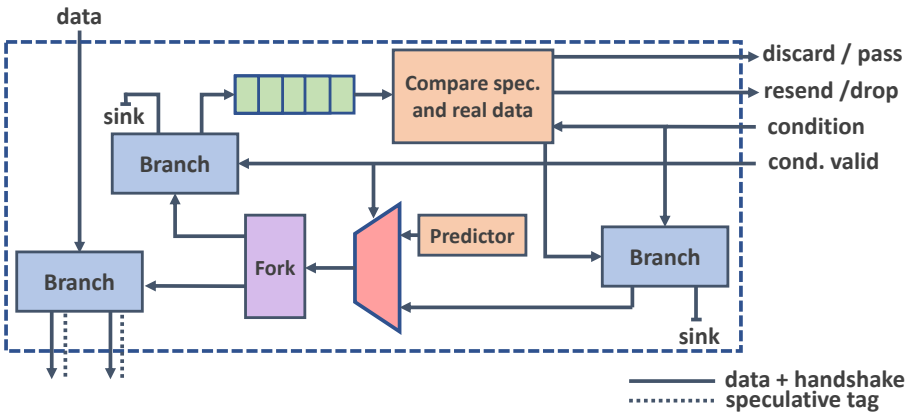


Figure 7.4 – Branch speculator. A branch speculator can speculate on the branch condition and output a speculative data token. It later on determines the correctness of a speculation and communicates this information to the save and commit units.

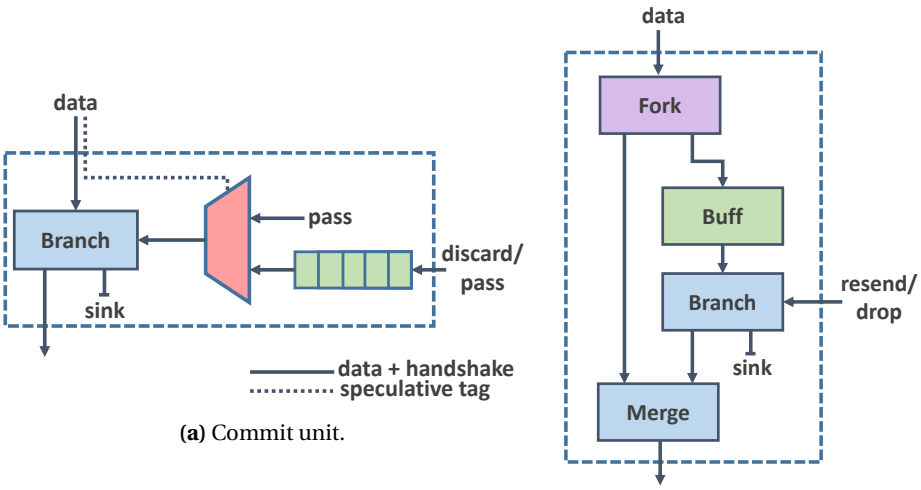
the real token (using the branch subcomponent on the right of Figure 7.4) or resend it into the circuit. The speculator can issue a token only when its basic block is active, otherwise, there is no guarantee that the real token will eventually arrive to confirm or cancel the speculation (this is easy for a branch speculator because the arrival of a data token is a guarantee that the condition will also arrive).

Initially, we will discuss the case where only one speculative token at a time is issued into the circuit—i.e., a new speculation cannot start before the previous one has been resolved. While this is not a problem for pieces of code that do not need to repeat (i.e., speculating on a branch of an *if-else* statement), it could easily result in suboptimal performance if the speculator is placed on a cyclic path (i.e., speculating on a loop termination condition). We will make necessary modifications to support multiple speculations from a single speculator in Section 7.6.

7.3.2 Commit Unit

All dataflow components, apart from the speculator, use a conservative firing rule: they produce tokens only once all of the required input operands become available. However, if one of the inputs of a dataflow component is a speculative token, the produced output token will become speculative as well—there is no guarantee that the computed value or the decision made by the component is correct until the speculation is resolved by the speculator. In case the speculation is incorrect, the component will output incorrect data or send a token in the wrong control flow direction: at some point, this misspeculated data will need to be discarded.

To this end, we use *commit* units that stall speculative tokens until they receive the corresponding decision from the speculator: in case the speculation is determined correct, the speculative tokens are converted into regular tokens and passed on to the rest of the circuit; otherwise, they are discarded by this unit. Any regular token that reaches the unit is unaffected and simply propagated through.



(a) Commit unit.

(b) Save unit.

Figure 7.5 – Components for speculation. The commit unit (Figure 7.5a) stalls speculative tokens until the correctness of the speculation has been determined. The save unit of Figure 7.5b saves tokens that might interact with speculative ones to be able to replay the computations in case of a misspeculation.

Figure 7.5a outlines the structure of the commit unit. Data enters the unit through an internal branch; depending on the value of the speculative bit, it is either directly passed on to the successor components (in case the data is nonspeculative), or stalled until the unit receives a decision from the speculator. Assuming that the data path from the speculator to this unit is long, the speculator might issue and resolve multiple speculations before the data tokens arrive at the commit unit. Hence, the unit contains a queue to save the decisions from the speculator if they arrive before the data. As the tokens arrive in order on both paths, the timing relations of the two paths cannot influence correctness: the commit unit will keep the first speculative piece of data on one path until the first confirmation or cancellation on the other path becomes available, and all tokens will be correctly matched. The output of the commit unit is always a regular nonspeculative token.

7.3.3 Save Unit

In case a speculation is determined incorrect, speculative tokens are discarded and speculated computations need to reexecute with the correct values. This means that each nonspeculative token which at some point interacts with a speculative token needs to be appropriately saved until the speculation is confirmed or canceled. To this end, we use *save* units which store the last token that passed through it until the speculator determines the correctness of the speculation. In case the speculator indicates that the speculation was correct or that it did not speculate on the saved values, the saved tokens are not needed and can be discarded—these values have already been correctly propagated through the circuit and their interactions with any token issued by the speculator produced correct results. On the other hand, if the speculation was

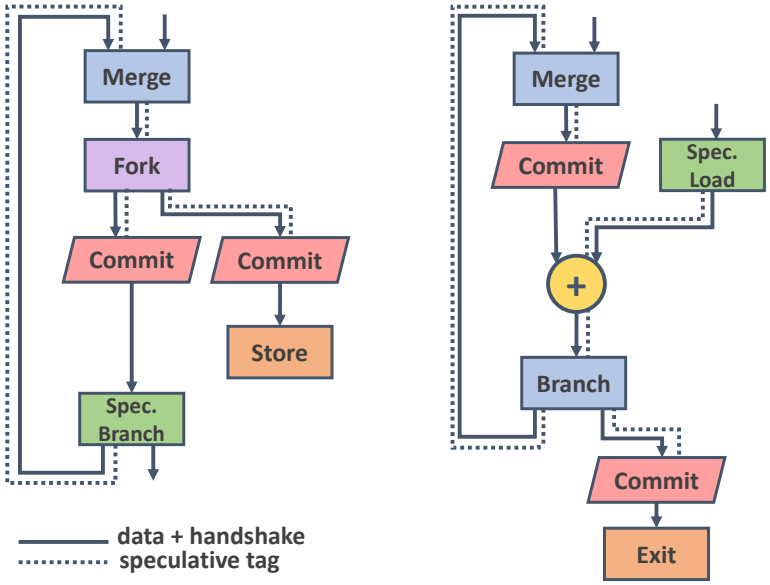


Figure 7.6 – Placing commit units. Our placement strategy ensures that memory is never modified by a speculative token, the program never terminates before speculation is resolved, and only nonspeculative values interact with units that might carry a speculative value.

incorrect, all save units need to reinsert their saved token into the circuit to repeat the previously miscalculated computations.

The save unit in Figure 7.5b takes a nonspeculative token as input and outputs a nonspeculative token. It requires only a single register for storing a token: for another token to arrive at the input (possible only if the unit is on a loop), the previous speculation must have been resolved and the old value inside the register has either already been reinserted into the circuit or determined unneeded and discarded through the branch.

Note that the *discard* and *resend* outputs of the speculator, connected to the commit and save units, respectively, are not equivalent: If a speculation does not occur, the save unit still kept a token which needs to be thrown away—the speculator must inform the unit when issuing a nonspeculative token. Commit units do not require any confirmation from the speculator to let the nonspeculative tokens pass.

7.4 Placing the Units

Every speculative region needs to be delimited with its own set of commit and save units: they ensure that misspeculated computations are appropriately squashed and replayed. This section shows where to place commit and save units into dataflow designs.

Every speculation needs to be resolved before terminating the program—that is, before a token reaches the exit node. Furthermore, only regular tokens can be used for modifying memory

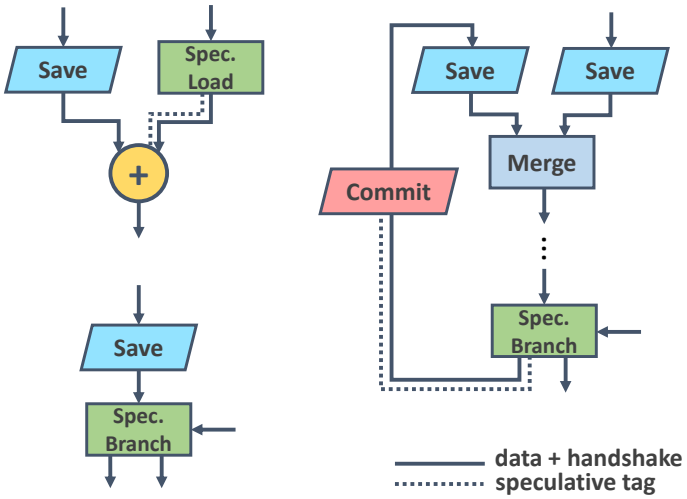


Figure 7.7 – Placing save units. Each token that interacts with a speculative token must be saved until the speculation is confirmed or canceled.

(assuming that writes cannot be reverted) or as inputs to the speculator (we will relax this constraint in Section 7.6). Therefore, we place a commit unit on each path of the dataflow graph which starts at the speculator and ends with the first of any of the following units encountered on the path: (1) an exit point of the graph; (2) the speculator or a unit carrying a speculative value; (3) a store unit. Figure 7.6 gives examples of correct placements of the commit unit. Placing more than one commit unit on a single path does not bring any benefit, as the first unit will always resolve the speculation. The commit units should be placed as far as possible from the speculator, as this allows speculating on more computations and therefore increases performance in case the speculation was correct.

A save unit is required whenever a regular token can interact with a speculative one, so the operations can reexecute in the case of a misspeculation. The following paths must contain a save unit: (1) each path from the start of the dataflow graph to any unit that could combine the token with a speculative value and (2) each cyclic path containing a speculator or any unit that could combine the token with speculative values. Since these cycles contain a commit unit (see previous section), the save unit must be placed after it—this ensures that only regular tokens enter the save unit, as any speculation will be previously resolved. Figure 7.7 shows examples of placing the save units. To maximize performance (i.e., a smaller number of correct computations to reexecute in case of a misspeculation) and minimize resource requirements (i.e., a smaller number of save units required), we place the save units as close as possible to the end of these paths (i.e., as close as possible to the paths carrying speculative tokens).

As already suggested, the dataflow circuit between a speculator and its commit units needs to carry data with a speculative tag. This modification requires only a minor change to standard dataflow units: it is simply one more bit of payload which is propagated or OR'ed from all inputs to make the output is speculative when any of the inputs is speculative, as depicted in Figure 7.8.

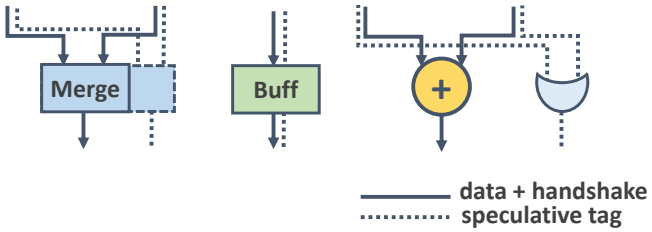


Figure 7.8 – Extending dataflow units with a speculative tag. In most cases (e.g., merge, buffer, fork), the tag is simply an additional bit propagated with the data. Units that combine multiple inputs (e.g., arithmetic operations) require an OR to make the output speculative when any of the inputs is speculative.

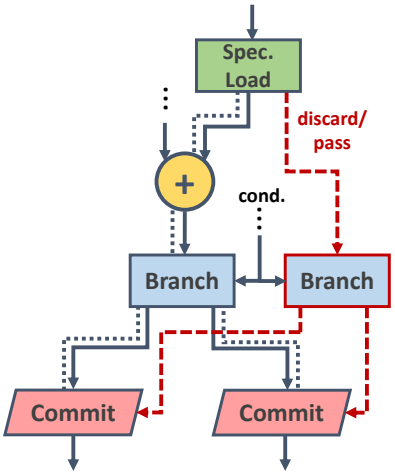


Figure 7.9 – Connecting the speculator to the commit units. The cancellation or confirmation from the speculator must be directed only to the commit unit which is on the path of the misspeculated token. Otherwise, another token could be discarded incorrectly: if both commit units in the figure were to receive a cancellation signal and the misspeculated token took the left branch, a correct token coming down the right branch would later be wrongly discarded.

7.5 Connecting the Units

When the speculator determines the correctness of a speculation, it needs to inform the appropriate save and commit units. We add a specialized handshake network for this purpose.

7.5.1 Connecting the Speculator to the Commit Unit

The speculator connects to the commit units through a specialized network and informs them whether to discard or propagate speculative tokens. However, sending the decision to all commit units would result in incorrect behavior. Consider the example in Figure 7.9: If a decision to discard the token due to a misspeculation is sent to both commit units, and the misspeculated token takes the left output of the branch, another token taking the right branch output later on would be incorrectly discarded. Therefore, the information from the speculator needs to be sent only to the units that were on the actual path taken by the speculative tokens. In such cases,

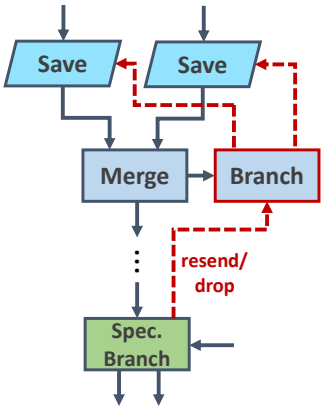


Figure 7.10 – Connecting the speculator to the save units. Any merge that is on the path from the save units to the speculator should memorize where tokens came from so that the speculator can send the correct resend or discard message to the appropriate save unit.

we place branches on the path connecting the speculator and the commit unit which receive the same conditions as the regular branches of the dataflow circuit. Whenever a speculative token passes, the branch in the specialized network will mimic the control flow decision taken by the data token and thus correctly direct the information from the speculator to one of the commit units.

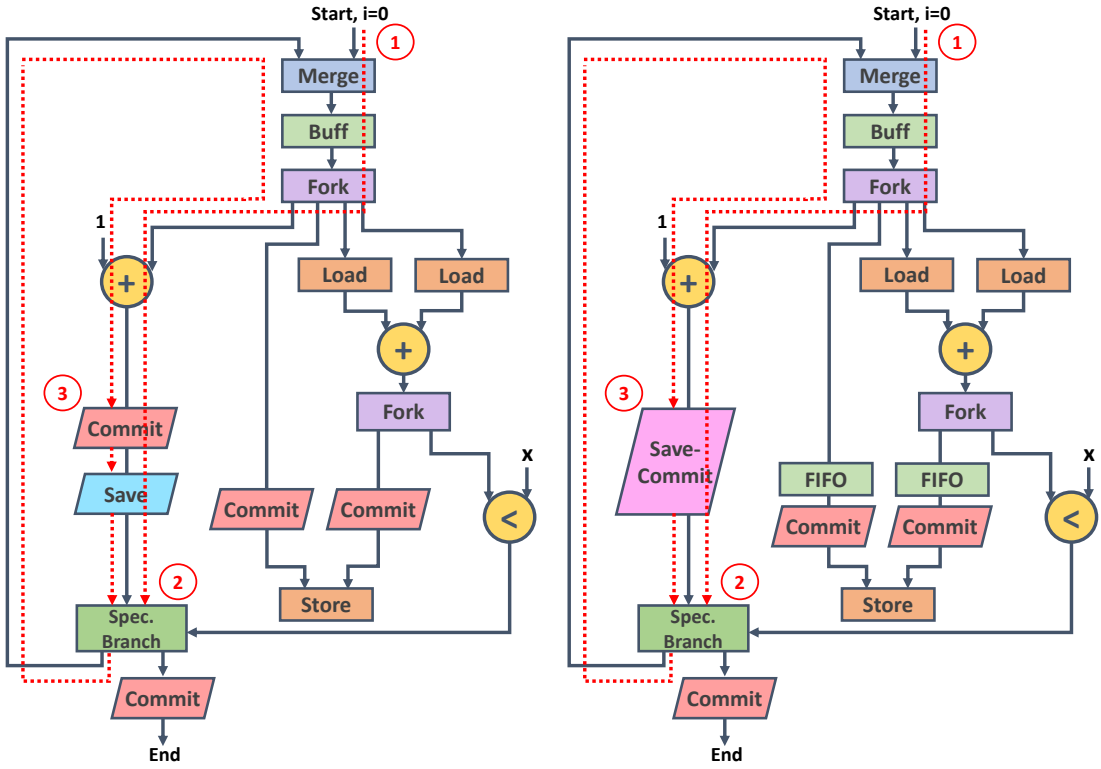
7.5.2 Connecting the Speculator to the Save Unit

The complementary problem arises when connecting the save units—only some of them hold tokens that need to be resent to the circuit. Consider the example in Figure 7.10, where the save units are placed before a merge node. If the speculation is determined incorrect, only one of the save units should reissue a token—however, there is nothing that can determine which of the two save units holds the direct predecessor (i.e., which token needs to be reissued). Therefore, merges that are on the path from the save units to the speculator need to remember which side a token came from. The speculator uses this information to correctly direct the confirmation/cancellation to the proper save unit. The dispatching is implemented in the specialized network as a branch which takes the speculator decision as data and the information from the original merge as the condition and forwards the decision accordingly.

7.6 Multiple Speculations from a Single Speculator

The approach described so far does not bring significant performance benefits when speculation occurs in a loop, as it requires us to conservatively wait for one speculation to end to be able to trigger a new one. This section discusses the modifications needed to increase loop parallelism.

In points where save and commit units meet, the approach described so far allowed a new token to enter the save unit only after the commit unit sent out a confirmed token. Thus,



(a) Speculating on a single value at a time.

(b) Speculating on multiple values.

Figure 7.11 – Enabling multiple speculations from a single speculator in the example from Figure 7.2. Our strategy from Section 7.3 results in suboptimal behavior when the speculation occurs in a loop: as Figure 7.11a shows, a token that is speculatively inserted into the loop will be stalled in the commit unit (point 3 in the figure) until the speculation is resolved, preventing the triggering of a new speculation. Merging the save and commit units on the loop into a single unit (Figure 7.11b) allows issuing a new speculative token before the previous speculation has been resolved.

all speculations through cyclic paths are sequentialized, which prevents us from achieving a high-throughput pipeline. Figure 7.11a shows the circuit from Figure 7.2 modified with the speculative units and the strategies described in the previous sections (note that the speculative tags are omitted for graphical simplicity). A nonspeculative token enters the merge through the starting point (labeled as point 1 in the figure), passes through the commit and save unit (as it is nonspeculative), and reaches the speculator (point 2). The speculator issues a speculative value back through the merge and into the commit unit (point 3), which stalls the token until the condition reaches the speculator and it informs the commit unit of the correctness of the speculation—only then does the token pass through to the speculator again, triggering the start of a new speculation.

7.6.1 Merging the Save and Commit Unit

Whenever a save and commit unit meet on a cyclic path, we can merge them into a single unit which allows issuing a speculative token even before the previous speculation has been

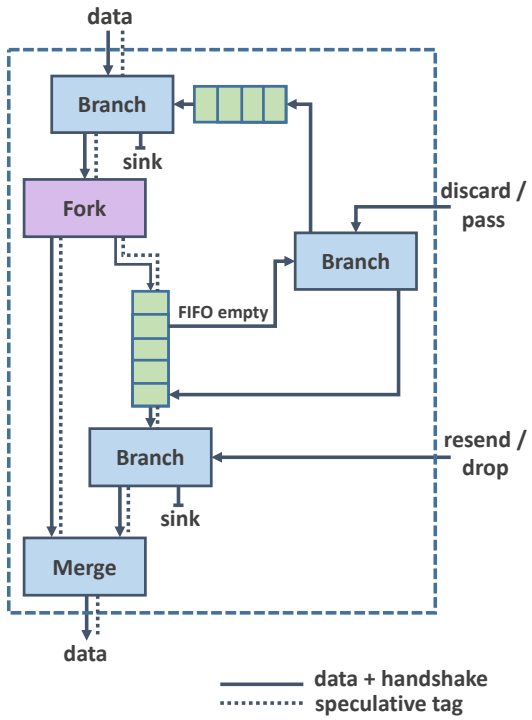


Figure 7.12 – The structure of the save-commit unit.

resolved. The *save-commit unit* (Figure 7.12) performs the combined functionality of both units: as a save unit, it issues regular tokens to restart computations or discards them when they are no longer needed; as a commit unit, it turns speculative tokens into regular ones or discards speculative tokens. However, unlike a regular commit unit, this unit will also let speculative tokens pass to the successors; it will save all the tokens, corresponding to regular or speculated data from multiple loop iterations, until they are no longer needed. We exploit the fact that the tokens are stored in the unit in order, as well as that the decisions arrive in order from the speculator—this allows us to easily match every decision to a token queued in this unit. The action of reissuing or discarding a token (usually performed by a save unit) will be applied on the oldest stored token which will, in both cases, be removed from the unit as it is no longer required: If the speculator informs the unit that a speculation was correct, the oldest token will be removed from the unit and its speculative successor will be transformed into a regular token. If the speculator sends a decision to discard a misspeculated token, the oldest speculative token will be discarded. The speculator will issue cancellations for each speculative token produced after the first misspeculation and each will discard one of the queued tokens. If the data tokens to cancel are not yet available, the cancellations are queued in a dedicated FIFO and the data is discarded as soon as it enters the unit.

Figure 7.11b shows the circuit of Figure 7.11a where the save and commit unit on the loop has been replaced with a combined save-commit unit. As before, the token enters through the merge (point 1 in the figure) and is sent to the speculator. The speculator issues a speculative token (point 2), which is stored in the save-commit unit (point 3), but also immediately propagated

to the speculator to trigger another speculation, hence finally resulting in the high-throughput pipeline achieving the lower schedule of Figure 7.1.

7.6.2 Connecting the Speculator to the Save-Commit Unit

There are two paths connecting the save-commit unit and the speculator, and both could contain control flow decisions: the one from the save-commit output to the speculator could contain merges (exactly like the path from the save unit to the speculator in Figure 7.10), and the one from the speculator to the save-commit input could contain branches (same as the paths to the commit units depicted in Figure 7.9). Therefore, as in the previous cases, our dedicated network for sending decisions to this unit will have to collect the control flow information from the original circuit to ensure that decision tokens are distributed the correct way. The principle is exactly the same as for connecting the speculator to the save and commit units; however, each control flow point will now have to hold multiple control flow decisions (as many as the save-commit unit can accommodate tokens). Whenever the speculator sends a decision, the oldest queued condition will be used and discarded. This ensures that every unit is correctly informed of the speculation.

7.7 Speculations from Multiple Speculators

The methodology discussed in the previous section describes a circuit with only one speculator issuing speculative tokens. Our approach could be easily extended to support multiple speculators in the design. The save and commit units and their placement strategy would be exactly the same; the only difference is that each speculative token would need to be tagged to keep track of the speculation origin—these tags would enable each commit unit to properly handle speculative tokens (i.e., each commit unit would consider as speculative only the tokens from the speculator it is connected to; all speculative tokens of a different origin would be treated as nonspeculative).

7.8 Evaluation

In this section, we evaluate our speculation technique by comparing static, dynamic, and speculative circuits. The statically scheduled baselines are obtained using Vivado HLS. We compare them with dynamic designs produced using the methodology described in Chapter 2, which results in nonspeculative circuits like that of Figure 7.2; we extend these circuits with the speculative units presented in this chapter to obtain circuits as in Figure 7.11b, which are our main results. Although our speculative methodology is perfectly general, in our examples we speculate on a single control flow decision using branch speculators from Figure 7.4. The speculators contain a static predictor that assumes the branch is taken whenever the input data becomes available. Each design contains as many speculators as there are variables which need to be speculatively issued to the successor basic block. All designs use identical floating-

point and integer arithmetic units and connect to the exact same RAM interface as the baseline designs from Vivado HLS. We use simulations in ModelSim [88] for functional verification and for measuring the loop initiation intervals. We synthesize the designs with Vivado to obtain the clock period and resource usage after placing and routing the designs.

7.8.1 Benchmarks

The designs that we consider in this section represent typical cases which can profit by branch prediction and where speculative execution should bring significant performance benefits over conservative, static scheduling. The benchmark loops are derived from real applications which can be found in literature [99].

- *While loop* is the kernel from Figure 7.1. The dynamic design results in the circuit of Figure 7.2 which we extend with speculative units to obtain the circuit of Figure 7.11b.
- *Backtrack* is the inner loop of the backtracking pass of the Bellman-Dijkstra-Viterbi algorithm. After labeling each state with the minimum cost to reach it, the backtracking pass looks for a unique set of edges that produce the global minimum. The states are traversed in a *for* loop which breaks when the predecessor state with the minimum cost is found. The *break* statement prevents loop pipelining, as the static tool starts a new loop iteration only after the break condition from the previous iteration has been determined false.
- *Subdiagonal* is an inner loop of a QL algorithm for determining the eigenvalues of a tridiagonal matrix. The loop looks for a single small subdiagonal element to split the matrix and contains a conditional *break* inside the loop body to return the correct subdiagonal index. As the condition for the return takes a long time to compute, it prevents static scheduling from efficiently pipelining the loop.
- *Fixed point* is an iteration method for finding the real roots of a function. It consists of a *while* loop which iterates through a sequence of improving approximate solutions until the desired degree of accuracy is achieved. Static scheduling postpones the start of a new iteration until the error computation from the previous iteration has been completed.
- *Newton-Raphson* is a hybrid algorithm of bisection and the Newton-Raphson method for finding the roots of a function. The hybrid algorithm takes a bisection step whenever Newton-Raphson would take the solution out of bounds and therefore improves the convergence properties of the algorithm over the standard Newton-Raphson method. The algorithm contains a *for* loop with an *if-else* statement to determine which of the two methods to use for a particular data point. Static predication is limited by the complex *if* condition and, as the next loop iteration requires the data computed in the current one, it must be scheduled for after the condition has been determined.

7.8.2 Results

Table 7.1 reports the timing and resource requirements of our benchmarks. The static scheduler constructs a conservative schedule which prevents almost any pipelining of these loops because

Table 7.1 – Timing and resource requirements of static, dynamic, and speculative circuits.

Benchmark	Design	II	CP (ns)	Time (μs)	Speedup	Slices	LUTs	FFs	DSPs
While loop	Static	11	3.7	37.4		130	270	436	2
	Dynamic	12	4.4	48.8	0.8×	129 (-1%)	353	511	2
	Speculative	~ 1	4.8	4.5	8.3×	186 (+43%)	486	582	2
Backtrack	Static	21	3.7	76.2		175	353	625	5
	Dynamic	22	3.5	75.6	1.0×	251 (+43%)	555	859	7
	Speculative	~ 1	5.1	5.1	14.9×	320 (+82%)	774	956	7
Subdiagonal	Static	17	3.6	60.0		164	342	591	5
	Dynamic	18	3.6	64.0	0.9×	179 (+9%)	424	611	5
	Speculative	~ 1	4.6	5.1	11.8×	233 (+42%)	559	650	5
Fixed point	Static	15	3.3	3.3		187	354	573	5
	Dynamic	17	3.3	3.8	0.9×	177 (-5%)	371	581	5
	Speculative	~ 6	3.8	1.6	2.1×	198 (+6%)	477	601	5
Newton-Raphson	Static	8	5.4	4.3		201	585	636	9
	Dynamic	10	5.0	5.1	0.8×	234 (+16%)	775	498	9
	Speculative	~ 1	5.5	0.6	7.2×	348 (+73%)	1181	603	9

it waits for the condition to be determined before starting a new loop iteration. Despite the flexibility of dataflow circuits, dynamic scheduling alone does not suffice to achieve high parallelism for the exact same reason as static scheduling: a new loop iteration is delayed until the previous decision has been determined, i.e., the branch waits for the condition token to arrive before propagating a data token backwards into the loop body. In contrast, the speculator in the final design issues speculative tokens into the loop as soon as the input data becomes available and enables achieving the ideal loop initiation interval. Note that the speculative initiation interval $\mathbf{II}_{spec}$ is, in fact, a weighted average of the value in case of good prediction and of that for a misprediction. For all circuits but *Newton-Raphson*, there is a single misprediction when the loop is exited and, therefore, the average II is for all practical purposes exactly the one in Table 7.1. Note that $\mathbf{II}_{spec} \approx 1$ for all benchmarks but *Fixed point*: in this case, the input data to the branch takes 6 cycles to compute, therefore limiting the maximum issue rate of speculative tokens. The resource increase and the longer critical path are due to the additional units for speculation and the FIFOs that we added to achieve maximum parallelism.

Although the table indicates $\mathbf{II}_{spec} \approx 1$ for *Newton-Raphson*, the situation is slightly different than in the other benchmarks: in this case, the misprediction is not an event happening only once per loop execution, but every time a bisection step is taken. The actual II is therefore data-dependent but still close to 1, as the bisection step is meant to be a relatively rare event. It is worth noting that our circuits do not have any additional penalty for misprediction other than incurring the longer latency of the corresponding dynamic nonspeculative circuit. Therefore, in general and to a first-order approximation (because we ignore the difference in critical path), our circuits would perform better than a static circuit whenever the prediction accuracy $p_{correct}$ is such that $p_{correct} \cdot \mathbf{II}_{spec\_opt} + (1 - p_{correct}) \cdot \mathbf{II}_{nonspec} < \mathbf{II}_{static}$. To put this in perspective using

```
int i = 0;
int s = 1;

for (i = 0; i < 12; i++){
    if (x[i]*s >= 1000)
        s+=1;
}
```

Figure 7.13 – Code used for the analysis of Section 7.8.3, qualitatively similar to the Newton-Raphson benchmark.

this example and again ignoring the CP difference, our circuit needs here only $p_{correct} > 22\%$ to perform better, and this branch prediction accuracy is massively below typical achievable rates.

7.8.3 Analysis

It is clear from Table 7.1 that the use of a dynamically scheduled paradigm has a nonnegligible cost in resources (we will discuss this overhead in detail in Chapter 9); the situation is only aggravated by the support for speculation. Although all our designs are Pareto optimal (and significantly faster than the baseline designs), it is worth looking closer into such results. As suggested before, predication is the way purely static scheduling methods can implement speculation (that is, by executing in parallel every possibility and selecting the right outcome later). It is usually viable when the number of predicated branches is small; thus, it is customarily used in the textbook case of if-conversion where only two short branches need to be followed for a very short period and are soon resolved. If resources are not strongly limited (that is, in the world of spatial computing as opposed to traditional VLIW compilation), one could explore an aggressive use of if-conversion where many branches are predicated at once—for example, with predication spanning multiple iterations of a loop body, as it would be required in some of our benchmarks. In this section, we want to explore how competitive our technique is against highly speculative, statically scheduled circuits beyond what our commercial tool produces.

We study here the code of Figure 7.13, which is qualitatively similar to our Newton-Raphson benchmark but stripped for clarity of everything except key operations. The naive version by Vivado HLS has II equal to 4 because of the loop-carried dependence on `s` and the multiplication (with a 4-cycle latency) in the condition which determines the new `s` (the conditional addition is predicated and executed in parallel). It is perfectly possible to restructure the code to perform aggressive if-conversion across basic blocks: every iteration spawns two branches corresponding to the new if condition, and this for each of the existing predicated branches; on the other hand, four cycles later, the computed condition resolves pairwise all open branches and halves them, leading to a steady state of in-flight branches. Assuming that the critical latencies are 1 for the addition and 4 for the multiplication, as it is the case for the operators used by Vivado HLS, achieving the II of 1 requires 16 parallel branches which compute `s` for every combination of the if conditions in the last four iterations and 8 branches computing the new conditions, also in turn depending on the conditions of the last three iterations. Essentially, the needed

Table 7.2 – Timing and resource requirements of the loop from Figure 7.13. The code given to the static tool was restructured to produce an aggressively-predicated schedule.

Design	II	CP (ns)	Time (μs)	Slice	LUT	FF	DSP
Static	1	5.7	0.1	1281	2088	5311	24
Dyn.	6	4.3	0.3	65	163	156	3
Spec.	2.3	5.3	0.2	154	481	301	3

computational resources to achieve II of 1 with a purely static schedule are 8 multipliers, 8 adders, and 8 comparators to execute all predicated branches in parallel.

Table 7.2 shows the comparison of the static, manually restructured code (to achieve a static schedule with II of 1), dynamic, and speculative design using a dataset which predicts correctly the condition in 75% of the cases. These results suggest that, although more speculation than what common HLS tools implement is possible, the cost can be very high (notice that the cost is exponential in $\mathbf{II}_{static}$, which is *only* 4 in this case). Clearly, our speculative circuit is Pareto-optimal compared to the aggressively-predicated static design. The area cost is due to the fundamental inability of statically scheduled circuits to revert some arbitrary computation and recompute it from scratch; a statically scheduled solution can only evaluate all possibilities at once and only when the number of possible outcomes is tractable. This is not the case in a situation we have not demonstrated here but is perfectly covered by our technique—the prediction of independence through memory of a load from all previous pending stores. In contrast, a dynamically scheduled, speculative circuit can simply execute the single most likely path and squash and recompute mistakenly predicted outcomes. In all fairness, this also implies a worsening of the execution time when almost-perfect predictions cannot be made, like in the present example, whereas the static solution has *exactly* II equal to 1, irrespective of predictability.

7.9 Conclusions

In this chapter, we presented a generic methodology to enable speculative execution in dataflow circuits and demonstrated that it can reap significant benefits in appropriate situations. Our simple and methodical approach to bring arbitrary forms of speculation to dataflow circuits mirrors out-of-order processors, where the same commit-or-squash-and-replay approach is at the heart of very successful speculative mechanisms. We have shown in Chapter 5 that dependences through memory are an important case where dynamic schedules are highly profitable: the next logical step would be to build a speculative load-store queue which executes speculatively loads before pending and unresolved stores, as in common processors; the generality of our speculation scheme would work unmodified for this important situation.

8 Related Work

In this chapter, we outline what others have done to circumvent some of the problems of statically scheduled HLS and we contrast our work with other dataflow-oriented approaches and optimizations.

8.1 High-Level Synthesis

As already discussed in Chapter 1, standard HLS tools borrow many optimizations techniques from VLIW processors to extract instruction-level parallelism: they rely on modulo scheduling [103, 17, 120, 34] to create pipelines with the best possible loop initiation intervals under the given clock and resource constraints and exploit aggressive code motion techniques to anticipate the execution of some operations before it is certain [92, 59, 60, 83]. Techniques for analyzing memory access patterns that we have discussed in Chapter 6, such as alias analysis and memory dependence analysis [5, 33], as well as optimizations to improve memory bandwidth, such as array partitioning and memory reuse [32], have been extensively studied in the context of static HLS. However, the conservatism of static scheduling hinders all these optimizations in the presence of complex control flow, statically undeterminable memory accesses, variable latencies, and loops with irregular bounds.

Recent advances in HLS have explored methods to overcome the conservatism in static scheduling and to remove the inability of HLS tools to handle dynamic events. Several techniques [2, 85] generate multiple schedules which are dynamically selected during runtime, once the values of all parameters are known; they rely on the capabilities of current HLS tools by replicating the source code and dynamically selecting which copy of the code needs to be executed. The drawback of these approaches is that they apply to only some very particular cases of dependences through memory; they are also affected by the area (or reconfiguration) overhead of synthesizing two or more versions of an accelerator and the cost of switching between them. Tan et al. [109] describe an approach called ElasticFlow to apply loop pipelining on a particular class of irregular loop nests with no inter-iteration dependences in the outer loops. In their approach, multiple pipeline instances of a dynamic-bound inner loop are scheduled to execute

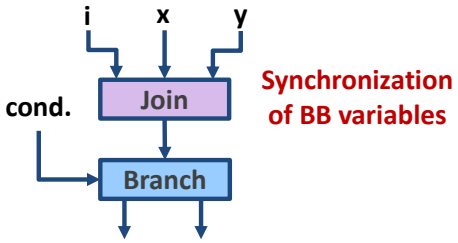


Figure 8.1 – BB variable synchronization by Huang et al. For all variables exiting a BB, Huang et al. [67] employ a single join, which limits look pipelining. In contrast, we employ an individual branch for each variable which enables tokens to exit the BB independently.

in parallel. Choi et al [31] et al. perform source-to-source HLS code transformations which rely on partial pipelining and unrolling to increase resource utilization in loops with variable bounds. Dai et al. [39] propose methods for pipeline flushing by performing static scheduling for multiple initiation intervals of the pipeline to resolve different possible resource collisions; they later developed application-specific dynamic hazard detection circuitry [40] and have shown the ability of speculation but with stringent constraints (i.e., the approach lacks generality in the ability to revert arbitrarily the state after failed predictions). Derrien et al. [42] provide support for control and memory speculation in statically scheduled datapaths; they keep track of the data from all speculated loop iterations which can be discarded and recomputed in case of misspeculation. However, if misspeculation occurs, the entire pipeline needs to flush and repeat, which may cause an increased misspeculation penalty in comparison to finer-grain speculation mechanisms. Nurvitadhi et al. [95] perform automatic pipelining, assuming that the datapath is already partitioned into pipeline stages. The underlying methodology in all these techniques is still based on static scheduling adapted to enable some level of dynamic behavior, which limits the achievable performance improvements only to some particular cases. We think that this body of recent work points to the importance of the ultimate solution to the limits of static scheduling: embracing general forms of dynamic scheduling.

8.2 Dynamic Scheduling in HLS

Different authors exploited latency-insensitive protocols [19, 37, 45] to construct synchronous and asynchronous dataflow circuits. Elastic circuits [37] are probably the best-studied form of latency insensitivity, but the original paradigm used in most of the papers by Cortadella and his coauthors is too restrictive for HLS. Several approaches [65, 25] extended the SELF protocol [37] with constructs similar to the branch and merge which we use in this work. Kam et al. [79] show the ability of elastic circuits to create dynamic pipelines, but do not provide generic transformations to create such circuits out of high-level descriptions. Efforts in the asynchronous domain, such as Balsa [44] and Haste/TiDE [93], applied syntax-driven approaches for mapping a program into a structure of handshake components [106]; a synchronous backend for Haste/TiDE has later been developed. Putnam et al. [100] also explored synthesizing dataflow-like circuits from high-level specifications. Townsend et al. [111] used a functional programming interme-

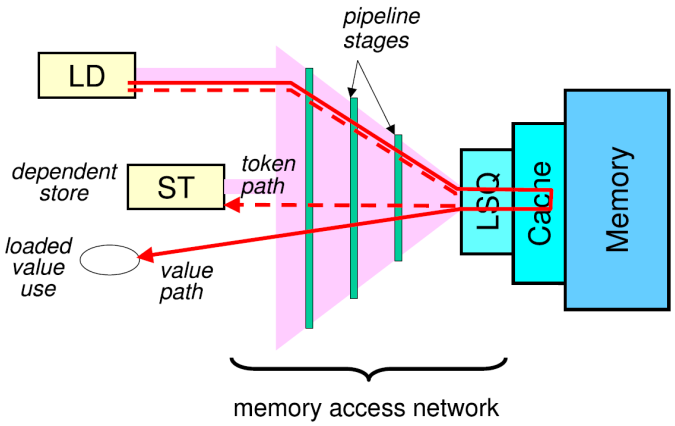


Figure 8.2 – Memory access synchronization by Budiu et al. The store in the figure may be dependent on the load; the two accesses are synchronized with a token edge which allows the store to issue only after the load has completed, hence preventing out-of-order execution when the two accesses are independent. Our LSQ effectively handles such cases by dynamically resolving dependences and synchronizing accesses only when needed. This figure is reprinted from the publication by Budiu et al. [12].

diate representation as a starting point for synthesizing dataflow networks. Dataflow circuits, with their handshake signals, bring to mind Bluespec and its firing rules [116]. However, all these approaches provide little information on some critical conversion aspects and features which are at the heart of this work; to our best knowledge, these approaches have never been contrasted to modern HLS tools.

The efforts closest to ours are the work by Huang et al. [67] and Budiu et al. [10, 9]. Huang et al. generated dataflow circuits from C code, to be mapped to a coarse-grain reconfigurable array [67]. Their circuit generation approach differs from ours in two aspects: (1) They use a single branch node at the output of each basic block, which forces them to synchronize all the basic block outputs (see Figure 8.1) and, consequently, prevents loop iterations from overlapping (i.e., loops are not pipelined). (2) Their approach does not employ an LSQ at the memory interface and, thus, all memory accesses which cannot be disambiguated at compile time need to be conservatively sequentialized ("The memory dependence is implemented by creating a lockstep between the corresponding [...] memory ports" [67]). Budiu et al. described a compiler for generating *asynchronous circuits* from C code [10, 9]. Although their final circuits are fundamentally different from ours (our circuits are *perfectly synchronous* and avoid the traditional difficulties associated with asynchronous designs), the generation strategy is similar to ours. Unfortunately, the exact methodology is never described in full detail and examples across different papers by the same authors do not seem perfectly consistent; although they also employ an LSQ to handle memory dependences and benefit from static analysis techniques to simplify it [11], their LSQ allocation policy is more conservative than what we described in Chapter 5: they serialize memory accesses whose dependences cannot be resolved statically ("we insert a token edge between two instructions only if their points-to sets overlap and they do not commute" [10]), as illustrated in Figure 8.2.

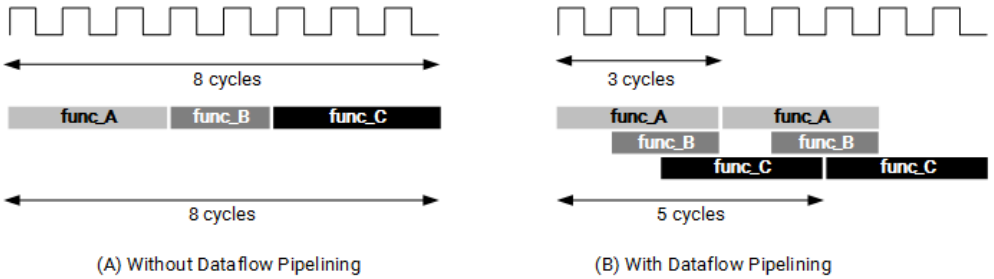


Figure 8.3 – Dataflow optimization in Vivado HLS. This optimization is applicable for coarse-grain tasks without conditionals and feedback, whereas we target fine-grain dataflow design of individual datapaths. This figure is reprinted from the Vivado HLS documentation [118].

Both the approach by Budiu et al. and by Huang et al. largely limit the benefits of dynamic scheduling; although Budiu et al. maintain the LSQ, Huang et al. omit it, most likely due to its seemingly limited value. Our LSQ, with its group allocation policy, enables spatial architectures to fully exploit memory access parallelism. Although Budiu et al. acknowledge the need to insert FIFOs of appropriate sizes to maximize throughput [10], they do not present a systematic methodology to perform this optimization; we tackle this problem in Chapter 3. Their HLS approach does not support resource sharing, which we include in Chapter 4. Finally, they failed to implement a generic framework for speculation due to "the difficulty of building a mechanism for squashing the computation on the wrong paths" [9], which is exactly what our scheme for discarding and replaying computations from Chapter 7 achieves.

8.3 Performance Optimizations of Dataflow Circuits

The schedules of dataflow circuits are not predetermined at compile time; their performance can be optimized using buffer retiming and recycling [20, 21], i.e., appropriate buffer insertion and sizing, which we exploit in this work. The timing properties of dataflow circuits can be analyzed using Petri net theory [102, 101, 16, 13]. We rely on this theory as well in our performance optimization strategy from Chapter 3. Several approaches in asynchronous dataflow design have explored slack matching, i.e., adding pipeline buffers to prevent stalls. Venkataramani et al. [115] present a heuristic to avoid performance bottlenecks by inserting buffers to balance reconverging paths in asynchronous circuits. Najibi et al. [91] describe slack matching for asynchronous circuits with conditional computation and communication, where the conditions correspond to different circuit operation modes; they employ a MILP based on Markov chains, a wide class of Petri nets, to balance asynchronous pipelines while reducing the number of slack-matching buffers. In contrast to these works, our model considers retiming and slack matching simultaneously—we target *synchronous* dataflow circuits, so the clock period must be optimized in conjunction with the throughput. Furthermore, our performance optimization model accepts generic control flow schemes that commonly appear in high-level languages (e.g., nested loops) and accounts for typical HLS features (e.g., pipelined computational units and if-convertible control flow).

Standard HLS tools support task-level pipelining (referred to as "dataflow optimization" [118]), which allows functions and loops to overlap to increase throughput and concurrency, as illustrated in Figure 8.3. The tasks are connected via channels, implemented as ping-pong buffers or FIFOs, that allow the consumer function or loop to start operation before the producer task has completed. The buffers are typically sized conservatively, so that they have the capacity to hold all data exchanged between tasks. Task-level pipelining is usually applicable only to tasks which do not have bypass, feedback, or conditionals between each other. In this thesis, we explore finer-grain dataflow design (i.e., scheduling individual loop and function datapaths); as we have demonstrated in our results, our approach successfully supports cyclic behavior and conditionals and is able to compute the required FIFO sizes even in those cases. Other HLS efforts also explore coarse-grain dataflow and the related buffer sizing problems. Cheng et al. [28] describe sequential programs as networks of processes in which hardware accelerators exchange data via FIFOs. To avoid deadlock, they analyze the static schedule of each accelerator and size the FIFOs accordingly. Yet, this approach does not always provide a global optimal solution for cases with multiple deadlock-causing cycles. Geilen et al. [51] use model checking to minimize buffer requirements while avoiding deadlock in coarser *Synchronous Dataflow Graphs* (SDFs). Our MILP model guarantees the absence of deadlock as well, while also maximizing throughput under a clock period constraint. Govindarajan et al. [54] target large-grain, multi-rate actor graphs and present an approach to minimize buffer storage while executing at the optimal computation rate. In contrast to our contribution, this approach does not consider constraining the clock period and it does not guarantee the optimality of the buffer placement.

8.4 Resource Optimizations of Dataflow Circuits

Several dataflow approaches support forms of resource sharing. In the context of HLS, Bluespec [7] allows the user to specify the appropriate control logic around a shared resource in a dataflow network using guarded atomic actions. Nielsen et al. [94] discuss the sharing of dataflow constructs in the context of the Balsa asynchronous hardware description language. Neither of these works addresses the correctness and performance aspects of sharing that we discuss in Chapter 4; furthermore, our approach automatically achieves the correct sharing logic without any user-given specifications.

Edwards et al. [45] present a nondeterministic sharing mechanism for dataflow circuits; yet, as we have illustrated before, such a mechanism is not sufficient to guarantee the absence of deadlock in dataflow circuits obtained out of imperative code. Cortadella et al. [35] describe sharing in elastic circuits and indicate the need to build a local scheduler to decide, at each clock cycle, which input can use the resource, both for avoiding unit starvation and for performance benefits. Similarly, Hansen et al. [62] employ a local, centralized FSM for every shared unit in their asynchronous pipelines to regulate the multiplexing of tokens at its inputs. However, both these approaches are applicable only to simple loops without conditionals, where a predetermined sequence of inputs can be encoded into a centralized scheduler. In contrast, our sharing method

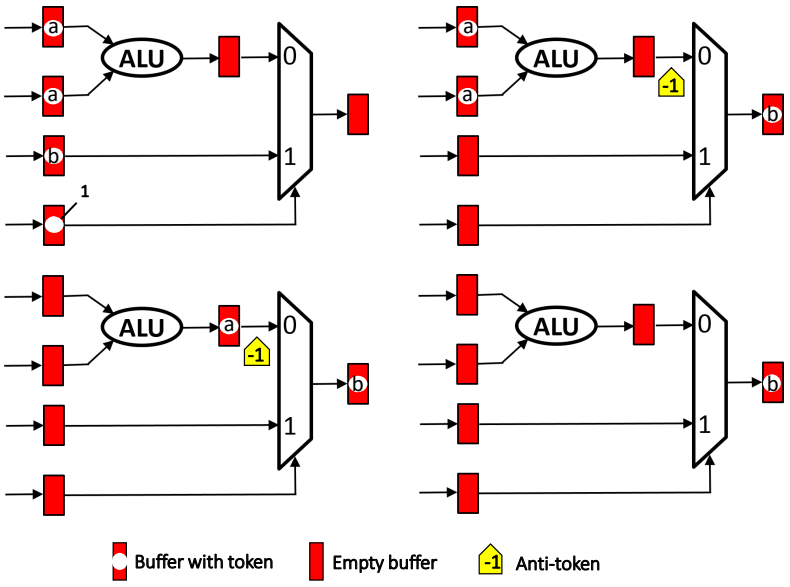


Figure 8.4 – Early evaluation with anti-tokens in elastic systems. This technique is applicable only for standard if-conversion, whereas our speculation approach supports generic forms of speculation. This figure is reprinted from the publication by Cortadella et al. [38].

applies to generic control flow construct obtained out of high-level code—we use a distributed network to control the token multiplexing dynamically, based on control flow outcomes.

8.5 Speculation in Dataflow Circuits

Latency-insensitive protocols offer the flexibility needed for true speculation [50]. Several latency-insensitive approaches [22, 36, 48] describe early evaluation—predicated execution based on special tokens which discard mispredicted data. An example of early evaluation is illustrated in Figure 8.4: the token produced by the ALU (annotated as *a* in the figure) is discarded by an *anti-token* as it reaches the multiplexer, which has already propagated the token from the other multiplexer input (i.e., token *b*) further, as determined by the condition value. However, these techniques are applicable only for standard if-conversion, which static HLS can handle—we also support it with our strategy from Section 3.4.4—but it does not cover the more general cases of speculation that we discuss in Chapter 7.

8.6 Computer Architecture

Out-of-order execution and speculation are typically employed in dynamically scheduled superscalar processors. The processor architecture community has studied LSQs for out-of-order processors for decades and has reported some innovations even in this millennium [96, 104, 97]. Although these previous designs have informed our implementation, they have not been specialized for spatial computing. Dynamically scheduled processors achieve generic forms of

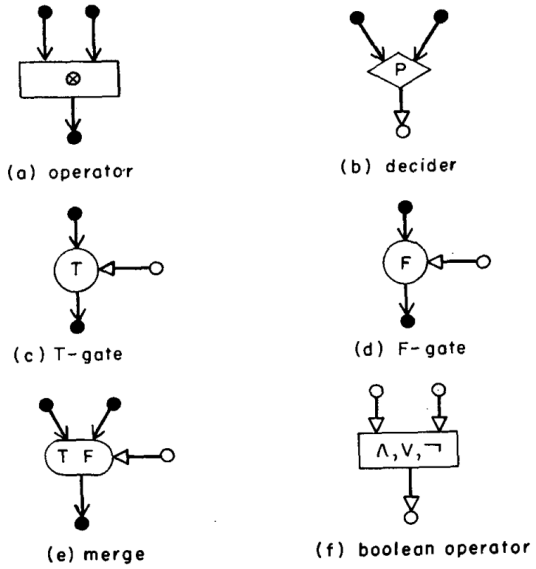


Figure 8.5 – Actor primitives of a dataflow processor. In contrast to standard processors, dataflow processors employ primitives similar to our dataflow units to steer data directly from one operation to another. This figure is reprinted from the publication by Dennis et al. [41].

speculation through register renaming, reordering buffers, and commit units [64]; our squash-and-replay technique for speculation in dataflow circuits, presented in Chapter 7, mirrors these mechanisms.

In contrast to standard processor architectures, dataflow processors [114, 41, 6] allow for direct exchange of data between instructions (i.e., data is sent from instruction to instruction without writing it back into a register file); instructions execute based on the availability of their input arguments and communicate using similar primitives as the ones employed in our work, as illustrated in Figure 8.5. More recently, EDGE architectures [14] have introduced control flow and more complex memory semantics to such processors, making them more compatible with imperative languages. Desikan et al. [43] describe a mechanism for load-store dependence speculation in the context of dataflow processors, but faced challenges in building a suitable speculation resolution network: their approach is based on sending the commit/discard decisions through the dataflow graph, so the traversal of these decisions delays the commits and therefore impedes performance. In contrast, we use a dedicated, fast network which enables speculative units to communicate directly and efficiently. Moreover, their speculation scheme requires version numbering and token tagging to handle out-of-order speculative bits—our circuit design strategy ensures that tokens traverse the graph in order, which simplifies our speculation mechanism.

9 A Complete Flow

In the previous chapters, we have shown how an arbitrary program described in a high-level language can be transformed into a dynamically scheduled, dataflow circuit. We described methods to implement pipelining and resource sharing; furthermore, we introduced mechanisms to handle out-of-order memory accesses and speculation. Each individual chapter has presented a number of results to demonstrate the effectiveness of the proposed technique; in this chapter, we describe our complete and automated compiler flow which supports the features presented in this thesis. We then summarize our most important results: a comparison of our HLS strategy with statically scheduled HLS.

9.1 Dynamatic HLS Compiler

Our dynamically scheduled HLS methodology is implemented in Dynamatic, our open-source HLS compiler [75]. The basic flow of Dynamatic is depicted in Figure 9.1.

Dynamatic takes as input C or C++ code and produces a synthesizable hardware description of the corresponding dataflow circuit. The first two steps of the flow, analysis and elaboration, preprocess the C files by prechecking code correctness, adding metainformation, and formatting it for the rest of the flow. The synthesis step relies on the LLVM compiler framework [86]: the *clang* frontend parses the C/C++ program and produces a static single assignment intermediate representation (LLVM IR), which is then optimized using standard LLVM transformation and analysis passes. The optimized IR is then given as input to a set of our custom passes.

The main pass adds dataflow units following the transformations described in Chapter 2 to produce a functionally correct dataflow circuit; other passes perform additional analysis and optimizations (e.g., memory access analysis to create the memory interfaces described in Chapter 6). The output is a dataflow graph in the form of a DOT netlist. The DOT netlist is then given as input to the optimizer which contains the buffer placement and resource sharing

This chapter is based on the work published at the *28th ACM/SIGDA International Symposium on Field Programmable Gate Arrays,* [75], 2020 and to appear in the *Circuits and Systems Magazine,* 2021 [76].

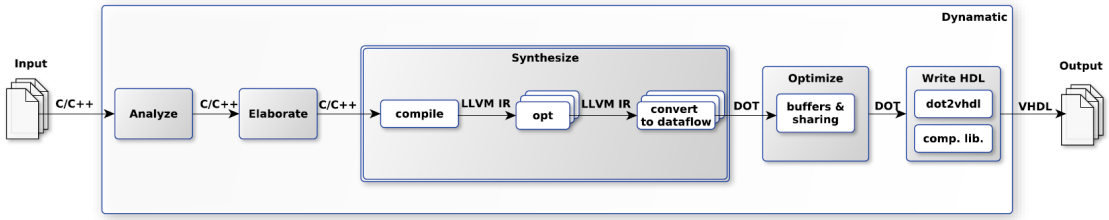


Figure 9.1 – Dynamatic HLS compiler: software-to-hardware flow.

tool—it uses a MILP solver [24] to find the optimal buffer placement and sizes for a user-defined clock period constraint, as indicated in Chapter 3, and decides on the functional units to share, as described in Chapter 4. This step produces an optimized DOT netlist. Finally, the DOT describing the dataflow circuit is converted into a VHDL netlist of dataflow units. This netlist, in conjunction with a predefined library of dataflow units and custom-generated memory interfaces (i.e., the LSQs described in Chapter 5), can be synthesized into an FPGA bitstream.

The entire Dynamatic toolchain, preinstalled in a virtual machine, is available on the Dynamatic website, together with tutorials and documentation about the tool: dynamatic.epfl.ch.

9.1.1 DOT Intermediate Representation

Dynamatic employs an intermediate representation for describing the synthesized dataflow circuits—the specification is based on the DOT language from Graphviz [55]. A dataflow circuit is represented by a digraph where each node corresponds to a unit and each edge corresponds to a channel. Units have input and output ports; channels are unidirectional and connect an output port from one unit to an input port of another unit. The units and channels can be annotated with attributes, which convey the information for different optimization steps (e.g., buffer placement) and VHDL generation. The DOT netlist can be converted into a graphical representation of the control/dataflow graph, with units grouped into basic blocks and connected via channels. A snippet of the DOT netlist of the histogram kernel and the control/dataflow graph corresponding to the netlist are shown in Figures 9.2 and 9.3.

9.1.2 VHDL Output

The VHDL netlist obtained in the final compilation step is a direct translation of the DOT netlist into an HDL description: all units are translated into VHDL unit instantiations and all channels are translated into VHDL signal connections. Every channel is represented with three signals: a data signal and a handshake pair. Apart from the VHDL netlist, the compiler generates application-specific memory components (e.g., LSQs).

The Dynamatic flow is device-independent—the produced VHDL netlist, as well as our HDL implementations of the dataflow units, are usable with any FPGA or for ASIC design. All dataflow units are fully parameterizable to arbitrary bitwidths and, as far as unit functionality permits, the

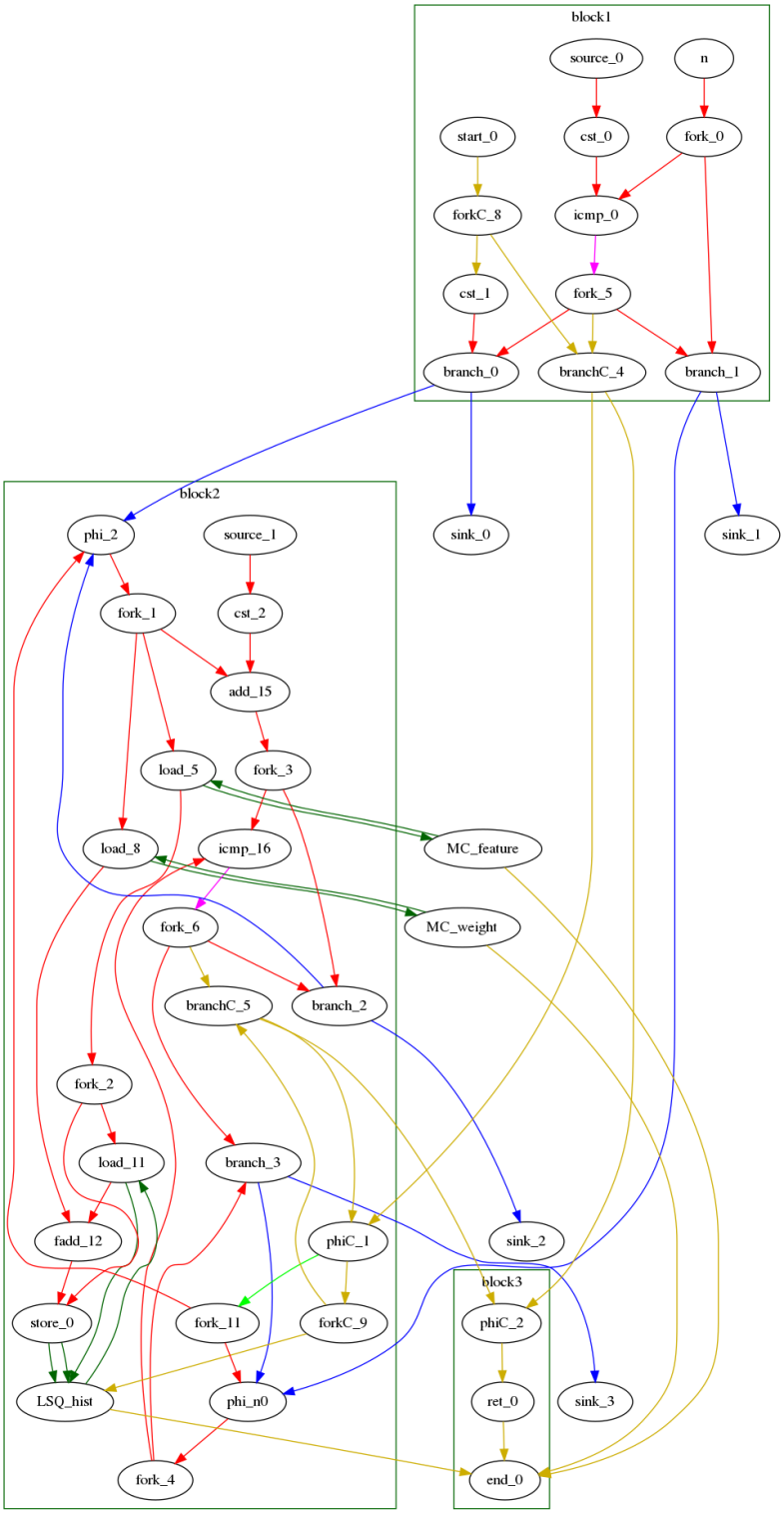


Figure 9.2 – Intermediate representation of a dataflow circuit in Dynamatic. The IR is organized as a CDFG of dataflow units grouped into BBs. The circuit corresponds to the histogram example.

```
"phi_2" [type = "Mux", bbID = 2, in = "in1?:1 in2:32 in3:32 ", out = "out1:32", delay=0.366];
"load_11" [type = "Operator", bbID = 2, op = "lsq_load_op", bbID= 2, portId= 0, in = "in1:32
in2:32", out = "out1:32 out2:32 ", delay=0.000, latency=2, II=1];
"fadd_12" [type = "Operator", bbID = 2, op = "fadd_op", in = "in1:32 in2:32 ", out = "out1:32 ",
delay=0.966, latency=10, II=1];
"store_0" [type = "Operator", bbID = 2, op = "lsq_store_op", bbID= 2, portId= 0, in = "in1:32
in2:32 ", out = "out1:32 out2:32", delay=0.000, latency=0, II=1];
"cst_2" [type = "Constant", bbID = 2, in = "in1:32", out = "out1:32", value = "0x00000001"];
"fork_0" [type = "Fork", bbID = 1, in = "in1:32", out = "out1:32 out2:32 "];
"branch_0" [type = "Branch", bbID = 1,  in = "in1:32 in2?:1", out = "out1+:32 out2-:32"];

"phi_2" -> "fork_1" [color = "red", from = "out1", to = "in1"];
"load_8" -> "fadd_12" [color = "red", from = "out1", to = "in2"];
"load_11" -> "fadd_12" [color = "red", from = "out1", to = "in1"];
"fadd_12" -> "store_0" [color = "red", from = "out1", to = "in1"];
"cst_2" -> "add_15" [color = "red", from = "out1", to = "in2"];
"add_15" -> "fork_3" [color = "red", from = "out1", to = "in1"];
```

Figure 9.3 – Snippet of the intermediate representation of the dataflow circuit in DOT format. The netlist contains a list of all dataflow units present in the design and specifies the channels (i.e., connections) between them. Units and channels are described with additional attributes.

number of inputs and outputs. The LSQ, implemented in Chisel, is parameterizable in depth, port count, as well as number and organization of connected BBs; Dynamatic automatically produces a configuration file used to generate custom LSQ instances for each application.

The arithmetic units employed by Dynamatic currently target the Kintex-7 family of Xilinx FPGAs; we extract them from the Vivado environment [119] and instantiate using Xilinx component libraries. All components employ a custom wrapper with handshake signals to be compatible with the rest of our dataflow units. Future releases of Dynamatic will extend the component library to target other device families and vendors as well.

9.1.3 Functional Verification

Dynamatic contains a verification framework which enables efficient software/hardware cosimulation, exactly like that supported by Vivado HLS [118]. The user can specify a testbench and a set of test vectors in C and the framework automatically produces the corresponding RTL testbench; it simulates the input C code and the output circuit and verifies that their outputs are identical. The RTL simulation is performed in ModelSim.

9.2 Evaluation

In this section, we compare the dynamically scheduled circuits produced by Dynamatic with statically scheduled circuits obtained from Vivado HLS. We give and overview of our methodology and benchmarks before presenting our results.

9.2.1 Methodology

To provide a fair comparison of our designs with those generated by Vivado HLS, we employ the same arithmetic units and we use the same RAMs for our designs. When our compiler cannot disambiguate memory accesses, we employ the LSQ in our designs and connect it to the RAM interface; otherwise, we connect the dataflow read/write ports to the RAM through a simple memory arbiter. In all Vivado designs, we apply the pipelining optimization directive.

The current release of Dynamatic supports all the features described in this thesis; the only exception is the insertion of speculative components into the dataflow circuit, described in Chapter 7. Hence, for the purpose of our experiments, we manually add the components for speculation to dataflow circuits which can benefit from this feature and resize the buffers to account for the increase in II caused by this modification. In addition, Dynamatic does not yet employ bitwidth analysis, a standard HLS optimization; we manually adapt the bitwidths of critical operations to match those employed by Vivado HLS. Furthermore, we manually specify the depth of the LSQ in benchmarks which require it (the need to use an LSQ is determined automatically by our memory analysis from Chapter 6).

We use the framework described in Section 9.1.3 for functional verification. We obtain the average loop initiation interval (II) from the simulation and the clock period (CP) from the post-routing timing analysis to calculate the total execution time. Placing and routing the designs using Vivado gives us the resource usage (i.e., the number of CLB slices, with the corresponding LUT and FF count, as well as the number of DSP units).

9.2.2 Benchmarks

The designs that we discuss in this section are simple kernels which represent typical cases where static scheduling is known to run into its fundamental limits, while dynamic scheduling should make a significant difference. We also consider two simple kernels where static scheduling is fully successful, to show that dynamic scheduling achieves virtually the same result with acceptable overheads. We already explored these benchmarks in the previous chapters to evaluate particular dataflow features; we here selectively include representative cases and provide a complete comparison with Vivado HLS.

- *Histogram* and *Matrix power* have memory access patterns that cannot be determined at compile time—there may be RAW dependences between the stores and the loads from the following iterations. We investigated these kernels in Chapters 3, 5, and 6.
- *If loop add* and *If loop mul* have a potential dependence across iterations which depends on the runtime-determined condition (i.e., the condition is determined based on data fetched from memory which is unknown during compilation). We introduced *If loop add* in Chapter 2; *If loop mul* is a variation of the previous kernel where we replace the conditional addition with a multiplication of the same variables, as mentioned in Chapter 3.

Table 9.1 – Timing comparison of dynamically scheduled circuits (our dataflow circuits) and statically scheduled circuits (Vivado HLS).

Benchmark	II_{avg}		CP (ns)		Exec. time (us)	
	STAT	DYN	STAT	DYN	STAT	DYN
Histogram	13.0	2.1	3.5	4.9	45.5	10.1
Matrix power	13.0	2.7	3.4	4.9	16.8	5.0
If loop add	10.0	1.1	3.2	5.0	32.0	5.5
If loop mul	7.0	1.1	3.2	5.2	22.4	5.5
FIR	1.0	1.0	2.9	3.6	2.9	3.6
MatVec	1.0	1.0	3.2	4.0	2.9	3.6
Backtrack	21.0	1.0	3.7	5.1	76.2	5.1
Newton-Raphson	8.0	1.0	5.4	5.5	4.3	0.6

- *Backtrack* and *Newton-Raphson* have long-latency, data-dependent conditions for starting a new loop iteration and could benefit from branch prediction; we looked into these kernels in Chapter 7.
- *FIR filter* and *MatVec* are regular kernels which do not have any memory or control dependences; we explored these kernels in Chapter 3.

9.2.3 Comparison with Static HLS

Tables 9.1 and 9.2 summarize the timing and resource results for all kernels and Figure 9.4 shows our results relative to those from Vivado HLS (results to the left or below the red square, which represents all Vivado designs, are better).

Timing. Avoiding conservative assumptions on memory and control dependences results in a significant improvement of the throughput and, consequently, execution time in all of the corresponding benchmarks. Note that the dynamic results are data-dependent: the best possible II is achieved when there are no dependences and the worst possible II when all neighboring iterations are dependent, as we explored in Section 3.6.6 and Table 3.4. It is interesting to note that even the worst-case dynamic II, reported in Table 3.4, is by 1 lower than the static II (i.e., equal to 12 instead of 13): when a load targets the same address as its preceding store, our LSQ directly forwards the store data to the load; in contrast, the static design is unaware of the accessed addresses, so it first stores the data and then reads it from memory.

The additional dataflow control logic (i.e., the merge, branch, fork, and join units which we insert into the design) and the LSQ (which is extremely sensitive to the number of queue entries, as we discussed in Chapter 5) impact the CP of the dataflow designs; the CP is typically higher than in the corresponding statically scheduled circuits. Although this timing overhead is quite tangible, it is still conspicuously small when compared to the potential improvement in II and, consequently, the net performance. In the *FIR* and *MatVec* benchmarks, static HLS techniques produce highly optimized pipelines because memory accesses can be disambiguated at compile

Table 9.2 – Resource comparison of dynamically scheduled circuits (our dataflow circuits) and statically scheduled circuits (Vivado HLS). The slice count for the kernels with the LSQ is shown as kernel slices + LSQ slices.

Benchmark	Slices		LUTs		FFs		DSPs	
	STAT	DYN	STAT	DYN	STAT	DYN	STAT	DYN
Histogram	129	220 + 1073	254	4294	510	2033	2	2
Matrix power	200	295 + 1020	340	4463	735	2055	5	5
If loop add	141	393	315	960	525	1318	2	4
If loop mul	177	348	334	892	655	1127	5	5
FIR	47	178	83	463	176	526	3	3
MatVec	63	298	129	843	221	631	3	3
Backtrack	175	320	353	774	625	956	5	7
Newton-Raphson	201	348	585	1181	636	603	9	9

time; although both the static and dynamic design achieve the ideal II of 1, these are the only cases where our results are Pareto-dominated by the static results due to the increase in CP.

Resource utilization. Table 9.2 contrasts the resource utilization of statically and dynamically scheduled circuits. The overhead in slices of the dynamic designs, notable across all benchmarks, is partially due to the control logic that the dataflow circuits contain and which allows them to achieve the latency-insensitivity that we desire. The overhead of the FIFOs that we introduced to increase throughput, as discussed in Chapter 3, is probably overblown by the simplicity of the examples with only a few functional units.

Vivado employs allocation and binding algorithms to share functional units among operators; sharing is possible without a performance penalty due to the low throughput which the static designs achieve. Since all the dynamic designs achieve high-throughput pipelines, sharing units is not possible without compromising throughput; we therefore allocate a new unit per operator in all the benchmarks we here explore, which contributes to the resource difference between the static and dynamic design. Consider, for instance, *If loop add*: in this example, our design requires two functional units to perform the addition and the subtraction whereas Vivado HLS time-multiplexes the same one (as evident from the DSP usage). By replacing one of the operations with a multiplication (i.e., *If loop mul*), we verified that the DSP count is now equal and the overall resource difference is smaller.

It is immediately visible from Figure 9.4 that the circuits requiring an out-of-order memory interface demand significant additional resources. It should be emphasized that the resource and timing overhead could be minimized by implementing the LSQs as hard-macros on FPGAs, in the same way as other memory hierarchy components might be in the future (e.g., caches and TLBs). In contrast to the expensive memory interface, our speculation mechanism does not introduce a significant area overhead, yet successfully accelerates all of the corresponding benchmarks by speculating on critical control decisions.

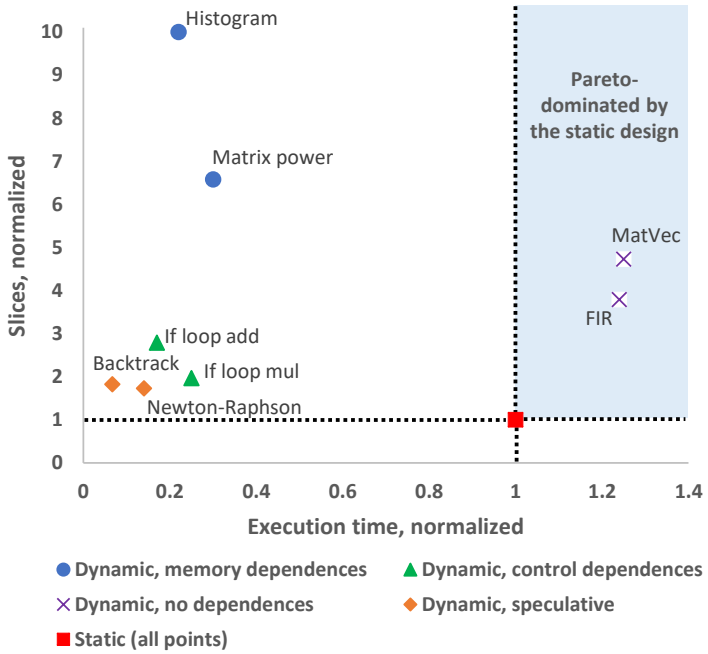


Figure 9.4 – Resource utilization and execution time of the dynamically scheduled designs, normalized to the corresponding static designs produced by Vivado HLS.

9.2.4 Conclusions

The results from Figure 9.4 show a clear tradeoff between statically and dynamically scheduled HLS: In regular applications, static HLS exploits all the available parallelism; dynamically scheduled HLS achieves the same, yet with additional CP and resource costs. On the other hand, in the presence of irregular program features, such as the ones outlined in Chapter 1 (i.e., unpredictable memory, control, or long-latency decisions), dynamically scheduled circuits exploit completely new optimization opportunities and achieve significant performance improvements at a resource investment. This tradeoff is completely in line with the experiences from computer architecture: the ability to make scheduling decisions dynamically requires complex mechanisms which inevitably cost resources and time. Although, due to this overhead, static scheduling is likely to remain the preferred solution for accelerating regular code, our dynamic approach opens doors to new, irregular applications and broader markets. In addition, the techniques described in this thesis are applicable outside of the scope of classic C-based HLS as well—we outline some possibilities in the following chapter.

10 New Avenues for Dynamic Scheduling

In this chapter, we discuss the possible future developments of our HLS approach. In Section 10.1, we outline relevant classes of applications which we have not yet extensively explored and where dynamic scheduling may bring significant benefits. In Section 10.2, we describe direct improvements of our HLS strategy. In addition to its contributions to HLS, this work opens doors to new research avenues and applications; we outline some of the most promising opportunities in Section 10.3. We then conclude this thesis in Section 10.4.

10.1 Application Domains for Dynamically Scheduled HLS

Our compilation process is generally applicable to any C/C++ program. In Chapter 9, we explored benchmarks that illustrate different representative cases and features discussed in this thesis: applications with irregular memory, irregular control flow, and long-latency control decisions. Benchmarks with these properties are not commonly present in standard HLS suites, which showcase primarily regular code that standard HLS can handle well. Thus, we primarily investigated kernels which are beyond the scope of these suites, yet representative of realistic software code and applications from different domains (e.g., signal processing, numerical analysis).

Although not evaluated in this thesis, dynamic scheduling is also beneficial for handling applications with variable loop bounds, such as sparse matrix computations. Sparsity is extensively exploited in various domains, from signal processing to AI (where sparse and irregular-shaped tensors are used to effectively implement large-scale applications such as object detection, recommendation systems, and language learning [107]). While standard HLS tools suffer in the presence of irregular loop bounds and memory access patterns (and, typically, require complex compilation techniques to efficiently pipeline applications which exhibit these properties), sparse kernels can be effortlessly compiled into a dataflow circuit without any additional analysis or transformation and pipelined directly with the strategy we presented in Chapter 3. Similarly, dataflow circuits are perfectly suited to tolerate variable memory latencies, an aspect that static HLS can handle only through complex code transformations (e.g., prefetching and access/exe-

cute decoupling to separate data access and address calculations from value computations for performance benefits [26, 61]). Dataflow circuits are latency-insensitive by construction and are able to adapt the schedule dynamically to variabilities in memory and operation latency without any additional effort from the programmer or the compiler.

Apart from further exploring the benefits of dynamic scheduling in the cases outlined above, the next step is to evaluate more complex applications (e.g., EEMBC [46] and CHStone [63] benchmarks) and to investigate the implications of the dataflow logic overheads on the resulting design quality (i.e., area, frequency, FPGA routability). Intuitively, our buffer placement strategy would ensure that even such benchmarks meet the frequency target; the heuristic presented in Chapter 3 would keep the compile times acceptably low even in complex applications by solving the buffering problem independently on sufficiently small application parts. As the applications (and, consequently, the resulting circuits) grow in size, place-and-route times and complexity may prevent an efficient FPGA implementation; these trends may be exaggerated in comparison to those observed in static HLS due to the additional complexity and area overheads of our circuit implementations. Yet, there are several optimization opportunities to explore which can minimize these overheads while still reaping the acceleration benefits of dynamic scheduling; we discuss them in the following sections.

10.2 Reducing the Costs of Dynamically Scheduled HLS

As discussed in Chapter 9, implementing dynamic scheduling typically comes at a notable resource and frequency penalty, especially in applications which require an LSQ. Although this cost is probably acceptable for irregular applications where performance significantly benefits from the dynamic properties of dataflow circuits, it is a clear overhead in regular DSP-like code that standard HLS targets, as our experimental results in Section 9.2 indicate. While it is expected that dynamic scheduling comes at a cost, it is also fair to note that statically scheduled HLS benefits from decades of research and optimizations which are yet to be explored in the dataflow context. We here evoke some of the most important areas which could directly improve our dynamically scheduled HLS.

10.2.1 Dataflow Graph Optimizations

Our compilation technique inserts dataflow units between operators; while some units only add control logic to the handshake network (e.g., a fork simply propagates the data from its input to its multiple outputs), other units impact the datapath as well (e.g., a merge has multiplexing logic to steer the appropriate input into a BB). Furthermore, in some units, data interacts with the handshake logic: for instance, the branch condition and the mux selection signal connect to both the forward- and to the backward-propagating handshake signals of these units, as their readiness and validity depend on the actual conditional outcomes. These situations may contribute to the increased critical path in comparison to the corresponding statically scheduled

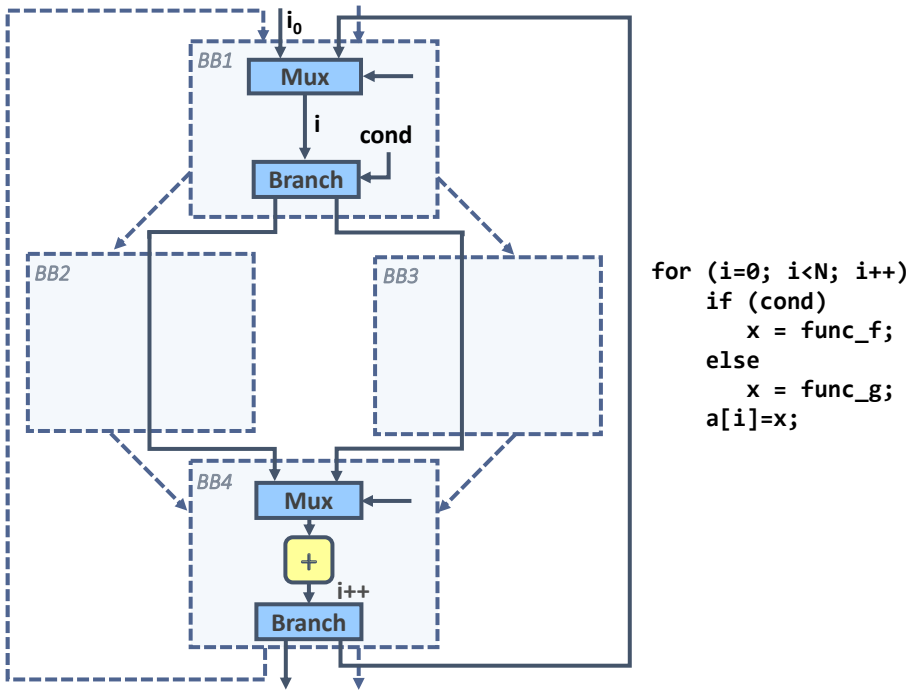


Figure 10.1 – Dataflow circuit optimizations. The dataflow circuit in the figure suffers in area and performance due to our circuit generation strategy, which forces all tokens to propagate through the BBs strictly following the control flow. The circuit could be simplified by sending the iterator directly from BB1 to BB4 and replacing the mux in BB1 with a merge.

circuit. Clearly, simplifying the dataflow network whenever possible would benefit circuit area and performance.

Bypassing BBs. Our circuit generation strategy propagates all tokens through the dataflow circuit strictly following the control flow. Although this approach guarantees correctness, as discussed in Section 2.3.2, it also adds complexity to the dataflow network by introducing dataflow units and buffers which statically scheduled circuits do not require. In addition, in certain cases, this approach may prevent throughput-critical tokens (e.g., a loop iterator) from propagating quickly through the loop. An example is shown in Figure 10.1: although the loop iterator is not used in BB2 and BB3, it is propagated into one of these blocks, as determined by condition *cond* in BB1, and only then sent to BB4 where it increments to start a new loop iteration. The start of a new iteration is not dependent on the value of *cond*; yet, if this condition takes multiple cycles to compute, the iterator will be stalled in BB1 and the start of a new iteration will be delayed—the loop II will increase, even though a perfect pipeline with an II of 1 is, in this example, possible.

Determining systematically when data can *bypass* certain BBs or CFG subgraphs would simplify the dataflow network and improve throughput, area, and critical path. In the example of Figure 10.1, sending the iterator directly from BB1 to BB4 would allow a new iteration to start on every consecutive cycle. The challenge in performing this optimization is to ensure that it does not compromise determinism nor the correctness of the circuit (by violating the correctness

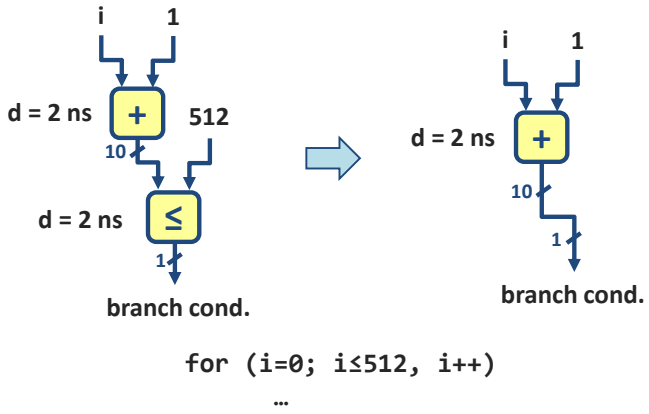


Figure 10.2 – Limitation of our static timing analysis. Our performance optimization technique is not aware of the transformations and optimizations that happen during synthesis, placement, and routing. In this example, our optimizer accounts for a delay of 4 ns between the adder and the comparator of the loop iterator (left), whereas these components are actually simplified during synthesis (right).

properties introduced in Chapter 2). Additionally, it should not interfere with the ability of the in-order control network to supply information on the correct BB ordering to the components that require it (e.g., the sharing logic from Chapter 4 and the LSQ from Chapter 5).

Replacing muxes with merges. Our current strategy employs a mux for every SSA *phi* node to guarantee that its inputs never reorder (see Section 2.3.3). However, not all *phi* nodes actually require them—there may be places where the inputs are guaranteed by circuit construction to arrive in order and the muxes could be replaced with simpler and cheaper merges. This is the case for the mux in BB1 of Figure 10.1: its left input is the initial iterator value, i_0; any other value of i is data-dependent on i_0 and is, therefore, guaranteed to arrive to the mux input later (i.e., only after i_0 has been sent into BB1). We could identify such situations using dataflow graph analysis to further simplify our circuits; the problem is similar to our memory analysis in Chapter 6 as it requires analyzing data dependences and producer-consumer relations across different paths of the control-flow graph.

10.2.2 Backend-Aware Transformations

Standard HLS techniques exploit a series of intermediate-level graph transformations to adapt the code to particular characteristics of the underlying hardware and to remove any unnecessary logic and constructs. A current disadvantage of our HLS flow is that it does not consider many particularities of FPGA synthesis, placement, and routing—we translate the generic LLVM IR directly into a dataflow circuit, while Vivado HLS uses custom passes to restructure this IR before translating it into hardware. Furthermore, our optimizations sometimes lack the necessary back-end information to achieve the best possible result. For instance, our performance optimizer relies on static timing information; if two independent operators have a delay greater than the clock period constraint, a buffer will be placed between them. However, in certain cases, these two operations may actually be simplified using logic synthesis and the resulting delays may

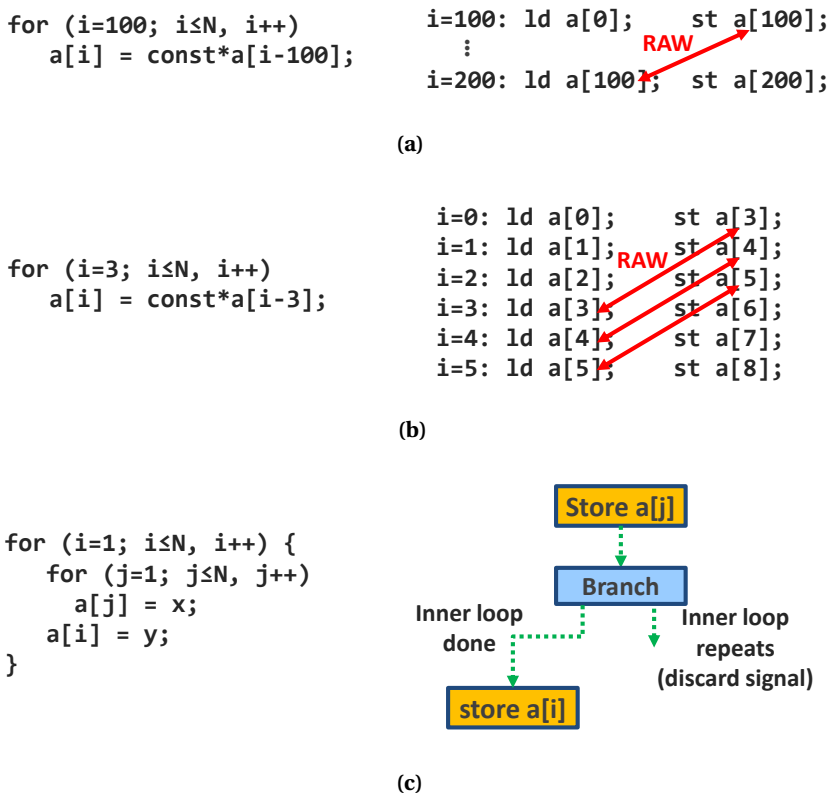


Figure 10.3 – Memory optimization opportunities. These examples illustrate cases where our LSQ could be simplified or removed to reduce the resource requirements of our circuits.

be substantially different than the ones we assume. Consider the example of the loop iterator in Figure 10.2: we account for the delays of the adder and the comparator unit, but the logic synthesizer in Vivado will simplify them into a single component: the highest bit of the addition is used directly as the branch condition (i.e., the condition to exit the loop is met as soon as the highest bit becomes equal to 1). In addition to the conservative delay calculation (i.e., based on the individual operator delays), our buffer placement may even prevent such optimizations by inserting buffers between these two operators; in contrast, Vivado HLS typically accounts for such optimization opportunities. These issues are not fundamental for dataflow design and our HLS flow could be further refined to account for such backend particularities.

10.2.3 Memory Interface Simplifications

As discussed earlier, the LSQs we employ usually cause a notable resource overhead. We have already explored techniques to simplify the memory interface in Chapter 6; it could be further optimized and customized for particular applications and memory access patterns.

Excluding the presence of hazards. As described in Chapter 6, our strategy places an LSQ whenever there is a RAW dependence between two instructions—if one could statically determine

that a hazard cannot occur due to a guaranteed cycle distance between the executions of the dependent accesses, an LSQ would not be needed. An example is shown in Figure 10.3a: due to the large iteration distance between the dependent accesses, one can easily determine that a hazard can never occur. Performing such optimizations may be more challenging in the presence of unpredictable events (e.g., control flow and variable latencies), which make it difficult to reason about timing relations between instructions.

LSQ simplifications. Our current LSQ implementation has generic comparison logic which checks all accesses for conflicts; this implementation could be replaced with simpler, application-specific logic which accounts for a particular memory access pattern. Consider, for instance, the example in Figure 10.3b: assuming that the load and the store in the figure are the only instructions connected to the LSQ, one could remove all address checks and use custom logic to synchronize the executions of dependent accesses (i.e., fixed LSQ entries); in this case, a load would be allowed to proceed only if the store from three iterations ago (i.e., with a *dependence distance* [11] of three, hence placed three entries 'ahead' in the LSQ) has already executed (or, similarly, the data from this store could be directly forwarded to the appropriate load).

Sequentializing accesses. In certain cases, forcing a particular ordering of accesses in the circuit may be possible without a significant performance penalty—serializing such accesses (e.g., by inserting additional token edges) instead of employing an LSQ could significantly reduce the resource requirements. For instance, the two stores in the nested loop in Figure 10.3c have WAW dependences between them and we, therefore, connect them to an LSQ. A favorable solution would be to delay the store of the outer loop until all stores from the inner loop have completed—sending a confirmation signal from the inner loop to this store would appropriately synchronize the accesses and remove the need to use an LSQ, without compromising the throughput of the innermost loop. These confirmation signals would need to be appropriately issued through the control flow graph, as the figure suggests: the branch sends to the store in the outer loop only the completion confirmation from the last store access of the inner loop.

LSQ sizing. Finally, as we have noted before, we have not done anything yet to automatically determine the optimal LSQ depth (i.e., the optimal number of LSQ entries). As suggested in Section 5.6, this parameter significantly impacts area, throughput, and critical path of the circuit; one could explore trade-offs between throughput and CP or look for a minimal queue depth which sustains the highest possible throughput. The complex problem of finding such a minimal depth depends on factors like circuit throughput (i.e., how often does the LSQ receive memory requests from the circuit), memory congestion (i.e., how long do the entries stay in the LSQ before they are issued to memory), and memory dependences (i.e., how long does it take to send data back into the circuit or to memory).

10.2.4 Partial Schedule Rigidification

One optimization aspect which is immediately manifest when looking at the circuits we generate is that we allow latency insensitivity through any unit and on any path. Although, in some cases,

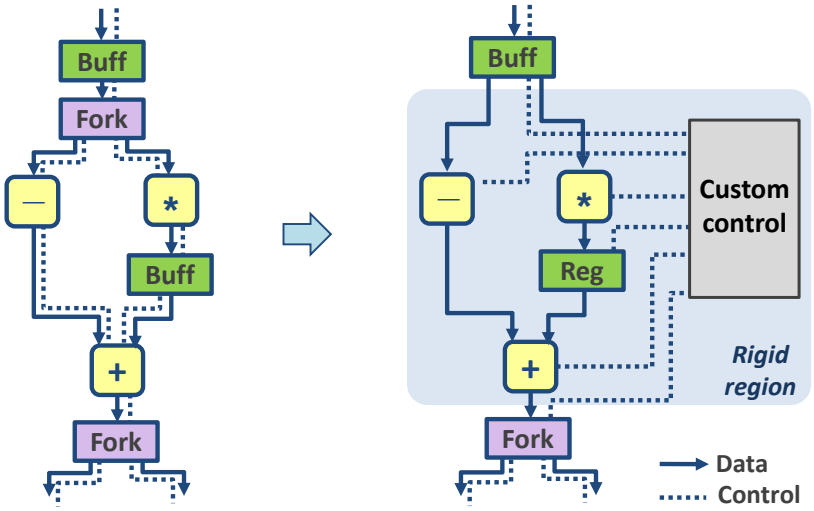


Figure 10.4 – Dataflow circuit rigidification. To simplify the dataflow circuit, the distributed handshake control logic could be replaced with customized control structures whenever dynamism is not required.

this is exactly the strength of our methodology and the reason for its superiority over standard HLS techniques, in many cases it is an expensive overkill: many computational paths may be constructed with fixed-latency units (ALUs, floating-point operators, etc.) and never really profit from the flexibility of dataflow computation. We already demonstrated the benefits of manually replacing functions or dataflow subgraphs with their static counterparts [27]; the next step is to develop optimizations which automatically rip off, under certain conditions, complex control paths of a dataflow circuit and replace them with simpler, customized control structures. One could see this as a *selective rigidification* of the schedule where dynamism is not really needed.

The challenge in performing rigidification is to automatically identify which units and paths may be rigid, without compromising performance or circuit correctness. We already exploited Petri net theory to obtain information on the flow of tokens through the dataflow graph—this information may be critical to identify units through which data always flows at a constant rate. These units do not require handshake logic—instead, they could be triggered using a local, predetermined scheduler which ensures that data is received and dispatched at appropriate time intervals. For instance, if it is certain that a buffer *always* holds a token for *exactly* two cycles, this buffer does not need to consider the handshake signal from its successor—it can simply be programmed to accept and output data on every second cycle. Multiple independent schedulers could eventually be merged into a single finite-state machine which would control the entire rigid portion of the circuit, as illustrated in Figure 10.4; this concept is similar to *relative scheduling* [81], where some operations (i.e., those in the rigid region) are statically scheduled relative to certain unknown delays (i.e., dynamic dataflow operations). The result would be a hybrid statically and dynamically scheduled circuit which enables the programmer to exploit the 'best of both worlds' [27], depending on the properties of the code: in regular applications, the final result would qualitatively correspond to a statically scheduled circuit;

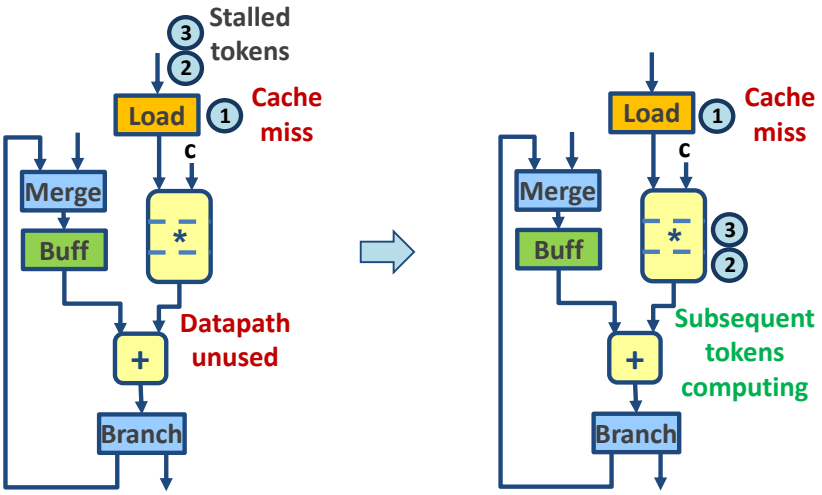


Figure 10.5 – Multithreaded execution. Instead of a single thread which always issues tokens in order into noncyclic paths, one could devise a system where tagged tokens execute out of order to increase performance and hardware utilization.

dynamism would remain only in places where it is actually required for performance benefits and at a significantly reduced area overhead.

10.3 Perspectives

Apart from enriching the capabilities of classic C-based HLS, our dynamic scheduling techniques can be useful for other avenues of research and applications—we here detail some promising directions and potential developments.

10.3.1 Multithreaded Execution

Our current approach targets standard sequential C-based synthesis: there is a single execution thread, i.e., a single token enters through the starting point, propagates through the BBs following the control flow, and exits through the final BB; pipelining is achieved by repeatedly issuing tokens in order into noncyclic paths. Yet, this type of circuit construction may result in limited parallelism and datapath usage in cases where pipelining is not possible (e.g., loop-carried dependences) or when multiple tokens on a noncyclic path are stalled waiting for a long-latency event related to some preceding token (e.g., a cache miss); an example of such situation is shown on the left of Figure 10.5.

Many standard HLS approaches support kernel replication to enable multiple parallel executions on independent copies of the datapath [30], hence fully exploiting the spatial parallelism of the device—the same is perfectly possible with dataflow circuits and could be achieved by high-level transformations (i.e., different input language or intermediate-level kernel replication) in dynamically scheduled HLS. In addition, some authors have looked into pipelining multiple

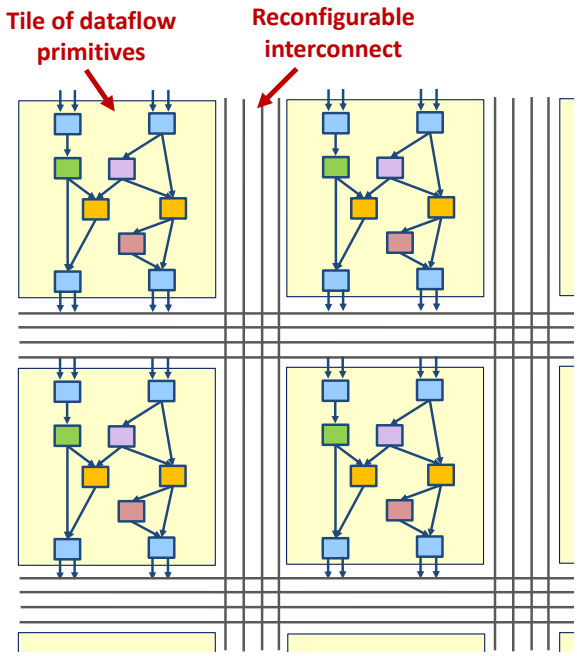


Figure 10.6 – A reconfigurable dataflow array, with tiles composed out of fixed interconnects of dataflow primitives. All connections carry data bundled with its control signals.

threads on a single kernel (i.e., allowing one thread to execute on the datapath before the previous thread has completed) [108]. Such *multithreading*, analogous to simultaneous multithreading in superscalar processors [113, 112], allows for maximal hardware reuse as resources can be shared among multiple threads. Just like superscalars, dataflow circuits are naturally suited to accommodate such behavior; it could be implemented by inserting multiple *tagged tokens* into the circuit and allowing them to reorder on both cyclic and noncyclic paths, as shown on the right of Figure 10.5. Enabling such multithreaded pipelines requires the creation of an efficient tagging mechanism which allows out-of-order execution wherever it is beneficial and reorders tokens appropriately when needed; similar mechanisms have been explored in dataflow architectures [43] and could be leveraged in this context as well. Furthermore, such a system would require additional guarantees on the absence of deadlock and appropriate buffering to accommodate the desired number of tokens on each dataflow path. Enabling dataflow kernel replication as well as multithreaded execution on a single kernel has the potential to significantly improve parallelism and resource utilization, hence bringing a completely new optimization dimension to dataflow design.

10.3.2 Reconfigurable Dataflow Architectures

So far, we have only attempted to map our dataflow circuits to standard FPGAs—a natural alternative to explore are coarser *reconfigurable arrays*, whose limited flexibility as well as word-oriented nature promise efficiency in area, timing, and energy [67]. The absence of a centralized controller and the systematic pairing of data with handshake signals makes dataflow circuits

particularly well-suited for such architectures: each array tile would be composed out of one or more dataflow primitives and the interconnect between tiles would carry data bundled with its control signals, as suggested in Figure 10.6.

One of the major challenges is to design an array which is structurally adequate for the computational patterns and interconnects which typically appear in dataflow circuits. Intuitively, circuits obtained from high-level code share some representative properties (e.g., BB organization with merges and branches at the inputs and outputs, respectively) which can be exploited to customize the array tiles. Our existing compilation flow can be used to translate a program into a functional netlist of hardware primitives; we would need to develop custom place-and-route techniques which exploit array-specific transformations and optimizations (e.g., breaking forks into sizes which the architecture supports, using physical components as wires to issue data efficiently from one component to another, array-specific buffering) to map these netlists onto the underlying architecture and to enable efficient architectural exploration.

10.3.3 Hardware Compilers and Dataflow Representations

Hardware accelerators are gaining popularity in different application domains; most rely on one or more domain-specific languages and the related HLS compilers use their own representations and transformations to obtain a hardware design. Although some compilation aspects are domain-specific (e.g., high-level transformations which convey domain-specific constructs to the tool), most compilers eventually rely on similar techniques (e.g., standard intermediate-level compiler transformations or low-level hardware constructs). Yet, different domains and frameworks offer only limited opportunities for reuse of transformations between them, which hinders the development of the tools and the quality of the resulting hardware.

Hence, developing core abstractions for hardware design that can be reused across compilation flows or unified into a single modular framework [89] would benefit the advancement and usability of HLS compilers. Such abstractions could then be appropriately manipulated and interchanged based on the application, high-level starting point, and the underlying hardware. Dataflow representations are extremely important in this context: they are suitable for modeling various processes of different granularities (i.e., from fine-grained dataflow to coarse-grain modules connected with handshake signals) and usable in different compilation stages (e.g., describing an application in software as an interconnect of dataflow modules or mapping dataflow graphs onto the corresponding hardware primitives). We therefore believe that the work described in this thesis is useful for developing a broader scope of HLS techniques which are beyond standard C-based HLS flows.

10.3.4 Formal Verification

HLS tools typically rely on functional verification of a particular circuit through hardware simulation and software/hardware cosimulation [118]. However, performing exhaustive hardware

simulations may become unfeasible or extremely time-consuming as designs increase in complexity. Furthermore, this approach provides no formal proof on the correctness of particular HLS compilation steps nor the general correctness of the resulting hardware modules—while this might not be a critical problem for FPGA designs, it prevents the adoption of HLS tools in domains where design iterations are significantly more expensive (e.g., ASICs) [33].

We have used SMV-based model checking to formally verify the correctness of each dataflow unit as well as some simple dataflow circuits. However, these tools are not efficient for verifying more complex circuits: as any data transfer in a dataflow circuit can occur at any point in time, the number of possible states is largely dependent on the number of dataflow units, buffering, and particular data and control outcomes. To further benefit from such tools, we should investigate methods to limit the state space and ensure their scalability; these techniques would allow us to verify critical properties and behaviors (e.g., does the circuit execution *always* terminate, for *any* input data?). In addition, a completely unexplored avenue is the formal verification of our HLS transformations, i.e., formally proving that our circuit generation strategy is *always* correct for any semantically correct C code. The ability to formally verify particular low-level compilation stages would benefit not only C-to-dataflow conversion, but hardware compilers and their transformations in general.

10.4 Final Remarks

Specialized hardware will profoundly impact computing in the next decades, as large-scale applications disrupt the market and their computational demands rapidly grow. HLS tools are set to play a key role in this computing shift, as they make hardware devices accessible to diverse users. However, these tools are relying on a paradigm which is conceptually identical to the problem of compilation for VLIW processors: generating good static circuits from high-level languages requires peculiar code restructuring algorithms (e.g., modulo scheduling), demands expert user interaction (e.g., pragmas and code refactoring), forces worst-case assumptions on important issues (e.g., memory and control dependences), and precludes key performance optimizations (e.g., general forms of speculative execution). Therefore, HLS tools today are limited only to particular market segments and are not able to satisfy the needs of broader application domains.

In this thesis, we described a dynamically scheduled form of HLS which produces dataflow circuits. Compared to a commercial HLS tool, the result is a different trade-off between performance and circuit complexity, much as superscalar processors represent a different trade-off compared to VLIW processors: When static HLS is able to exploit the maximum parallelism available, our technique achieves similar results with some degradation in both cycle time and resources. When static HLS misses some key performance optimization opportunities, our circuits seize them by reordering memory accesses, dynamically resolving control dependences, and speculating on critical control decisions; dynamic HLS achieves significant performance improvements with the investment of more resources.

Apart from extending the usability of HLS to new application domains (i.e., irregular and general-purpose software applications), our HLS approach also leads way to building new classes of architectures (i.e., coarser reconfigurable arrays) which may be able to overcome some of the limitations of traditional FPGAs. In addition to its role in reconfigurable computing, this work promises to impact completely new markets: HLS tools today are largely restricted only to the FPGA domain because of their unreliability and unpredictability in obtaining good results; the ability of our approach to achieve acceptable performance without extensive experimentation and trial-and-error may contribute to the acceptance of HLS in other domains as well, such as ASIC design. Therefore, this avenue of HLS has the potential to open new doors for specialized computing and hardware compilers, to advance the usability of various hardware platforms, and to have a real-world impact on critical applications.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Company, first edition, 1986.

[2] Mythri Alle, Antoine Morvan, and Steven Derrien. Runtime dependency analysis for loop pipelining in high-level synthesis. In *Proceedings of the 50th Design Automation Conference*, pages 1–10, Austin, Tex., June 2013.

[3] Amazon.com, Inc. *Amazon EC2 F1 Instances*, 2017.

[4] Andrew W. Appel and Maia Ginsburg. *Modern Compiler Implementation in C*. Cambridge University Press, first edition, 1998.

[5] J. R. Appel and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann, first edition, 2001.

[6] Arvind and R. S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3):300–318, March 1990.

[7] Arvind, Rishiyur S. Nikhil, Daniel L. Rosenband, and Nirav Dave. High-level synthesis: an essential ingredient for designing complex ASICs. In *Proceedings of the International Conference on Computer-Aided Design*, pages 775–82, San Jose, Calif., November 2004.

[8] David F. Bacon, Rodric Rabbah, and Sunil Shukla. FPGA programming for the masses. *Communications of the ACM*, 54(4):56–63, April 2013.

[9] Mihai Budiu, Pedro V. Artigas, and Seth Copen Goldstein. Dataflow: A complement to superscalar. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pages 177–86, Austin, Tex., March 2005.

[10] Mihai Budiu and Seth Copen Goldstein. Pegasus: An efficient intermediate representation. Technical Report CMU-CS-02-107, Carnegie Mellon University, May 2002.

[11] Mihai Budiu and Seth Copen Goldstein. Optimizing memory accesses for spatial computation. In *Proceedings of the 1st International ACM/IEEE Symposium on Code Generation and Optimization*, pages 216–27, San Francisco, Calif., March 2003.

[12] Mihai Budiu, Girish Venkataramani, Tiberiu Chelcea, and Seth Copen Goldstein. Spatial computation. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 14–26, Boston, MA, October 2004.

[13] Dmitry Bufistov, Jordi Cortadella, Mike Kishinevsky, and Sachin Sapatnekar. A general model for performance optimization of sequential systems. In *Proceedings of the International Conference on Computer-Aided Design*, pages 362–69, San Jose, Calif., November 2007.

[14] Doug Burger, Stephen W. Keckler, Kathryn S. McKinley, Mike Dahlin, Lizy K. John, Calvin Lin, Charles R. Moore, James Burrill, Robert G. McDonald, and William Yoder. Scaling to the end of silicon with EDGE architectures. *IEEE Computer*, 37(7):44–55, July 2004.

[15] Peter J. Cameron. *Combinatorics: topics, techniques, algorithms*. Cambridge University Press, first edition, 1994.

[16] J. Campos, G. Chiola, J. M. Colom, and M. Silva. Properties and performance bounds for timed marked graphs. *"IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications"*, 39(5):386–401, May 1992.

[17] Andrew Canis, Stephen D. Brown, and Jason H. Anderson. Modulo SDC scheduling with recurrence minimization in high-level synthesis. In *Proceedings of the 23rd International Conference on Field-Programmable Logic and Applications*, pages 1–8, Munich, September 2014.

[18] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D. Brown, and Jason H. Anderson. LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems. *ACM Transactions on Embedded Computing Systems*, 13(2):24:1–24:27, September 2013.

[19] Luca P. Carloni, Kenneth L. McMillan, and Alberto L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-20(9):1059–76, September 2001.

[20] Luca P. Carloni and Alberto L. Sangiovanni-Vincentelli. Performance analysis and optimization of latency insensitive systems. In *Proceedings of the 37th Design Automation Conference*, pages 361–67, Los Angeles, Calif., June 2000.

[21] Luca P. Carloni and Alberto L. Sangiovanni-Vincentelli. Combining retiming and recycling to optimize the performance of synchronous circuits. In *16th Symposium on Integrated Circuits and System Design*, pages 47–52, Sao Paolo, September 2003.

[22] Mario R. Casu and Luca Macchiarulo. Adaptive latency insensitive protocols and elastic circuits with early evaluation: A comparative analysis. *Electronic Notes in Theoretical Computer Science*, 245:35–50, August 2009.

[23] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A cloud-scale acceleration architecture. In *Proceedings of the 49th International Symposium on Microarchitecture*, pages 1–13, Taipei, Taiwan, October 2016.

[24] CBC mixed-integer linear programming solver. *https://github.com/coin-or/Cbc*.

[25] Satrajit Chatterjee, Michael Kishinevsky, and Umit Y. Ogras. xMAS: Quick formal modeling of communication fabrics to enable verification. *IEEE Design & Test of Computers*, 29(3):80–88, June 2012.

[26] Tao Chen and G. Edward Suh. Efficient data supply for hardware accelerators with prefetching and access/execute decoupling. In *Proceedings of the 49th International Symposium on Microarchitecture*, pages 1–12, Taipei, October 2016.

[27] Jianyi Cheng, Lana Josipović, George A. Constantinides, Paolo Ienne, and John Wickerson. Combining dynamic & static scheduling in high-level synthesis. In *Proceedings of the 28th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 288–98, Seaside, Calif., February 2020.

[28] Shaoyi Cheng and John Wawrzynek. Synthesis of statically analyzable accelerator networks from sequential programs. In *Proceedings of the International Conference on Computer-Aided Design*, pages 126–33, Austin, Tex., November 2016.

[29] Derek Chiou. Intel acquires Altera: How will the world of FPGAs be affected? In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, page 148, Monterey, Calif., February 2016.

[30] Jongsok Choi, Stephen Brown, and Jason Anderson. From software threads to parallel hardware in high-level synthesis for FPGAs. In *Proceedings of the IEEE International Conference on Field Programmable Technology*, pages 270–77, Kyoto, December 2013.

[31] Young-kyu Choi and Jason Cong. HLS-based optimization and design space exploration for applications with variable loop bounds. In *Proceedings of the 37th International Conference on Computer-Aided Design*, pages 1–8, San Diego, Calif., November 2018.

[32] Jason Cong, Wei Jiang, Bin Liu, and Yi Zou. Automatic memory partitioning and scheduling for throughput and power optimization. In *Proceedings of the International Conference on Computer-Aided Design*, pages 697–704, San Jose, Calif., November 2009.

[33] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–91, April 2011.

[34] Jason Cong and Zhiru Zhang. An efficient and versatile scheduling algorithm based on SDC formulation. In *Proceedings of the 43rd Design Automation Conference*, pages 433–38, San Francisco, Calif., July 2006.

[35] Jordi Cortadella, Marc Galceran-Oms, and Mike Kishinevsky. Elastic systems. In *Proceedings of the 10th ACM/IEEE International Conference on Formal Methods and Models for Codesign*, pages 149–58, July 2010.

[36] Jordi Cortadella and Mike Kishinevsky. Synchronous elastic circuits with early evaluation and token counterflow. In *Proceedings of the 44th Design Automation Conference*, pages 416–19, San Diego, Calif., June 2007.

[37] Jordi Cortadella, Mike Kishinevsky, and Bill Grundmann. Synthesis of synchronous elastic architectures. In *Proceedings of the 43rd Design Automation Conference*, pages 657–62, San Francisco, Calif., July 2006.

[38] Jordi Cortadella, Marc Galceran Oms, Michael Kishinevsky, and Sachin S. Sapatnekar. RTL synthesis: From logic synthesis to automatic pipelining. *Proceedings of the IEEE*, 103(11):2061–75, November 2015.

[39] Steve Dai, Mingxing Tan, Kecheng Hao, and Zhiru Zhang. Flushing-enabled loop pipelining for high-level synthesis. In *Proceedings of the 51st Design Automation Conference*, pages 1–6, San Francisco, Calif., June 2014.

[40] Steve Dai, Ritchie Zhao, Gai Liu, Shreesha Srinath, Udit Gupta, Christopher Batten, and Zhiru Zhang. Dynamic hazard resolution for pipelining irregular loops in high-level synthesis. In *Proceedings of the 25th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 189–94, Monterey, Calif., February 2017.

[41] Jack B. Dennis and David P. Misunas. A preliminary architecture for a basic data-flow processor. *ACM Computer Architecture News*, 3(4):126–32, December 1974.

[42] Steven Derrien, Thibaut Marty, Simon Rokicki, and Tomofumi Yuki. Toward speculative loop pipelining for high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020. To appear.

[43] Rajagopalan Desikan, Simha Sethumadhavan, Doug Burger, and Stephen W. Keckler. Scalable selective re-execution for EDGE architectures. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 120–32, Boston, MA, October 2004.

[44] Doug Edwards and Andrew Bardsley. Balsa: An asynchronous hardware synthesis language. *The Computer Journal*, 45(1):12–18, January 2002.

[45] Stephen A. Edwards, Richard Townsend, and Martha A. Kim. Compositional dataflow circuits. In *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design*, pages 175–84, Vienna, September 2017.

[46] Embedded Microprocessor Benchmark Consortium. *https://www.eembc.org/*, 2020.

[47] Michael Fingeroff. *High-Level Synthesis Blue Book*. Xlibris Corporation, first edition, 2010.

[48] Hagen Gädke and Andreas Koch. Accelerating speculative execution in high-level synthesis with cancel tokens. In *International Workshop on Applied Reconfigurable Computing*, pages 185–95, Berlin, March 2008.

[49] Brian Gaide, Dinesh Gaitonde, Chirag Ravishankar, and Trevor Bauer. Xilinx adaptive compute acceleration platform: Versal architecture. In *Proceedings of the 27th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 84–93, Seaside, Calif., February 2019.

[50] Marc Galceran-Oms, Jordi Cortadella, and Mike Kishinevsky. Speculation in elastic systems. In *Proceedings of the 46th Design Automation Conference*, pages 292–95, San Francisco, Calif., July 2009.

[51] Marc Geilen, Twan Basten, and Sander Stuijk. Minimising buffer requirements of synchronous dataflow graphs with model checking. In *Proceedings of the 42nd Design Automation Conference*, pages 819–24, Anaheim, Calif., June 2005.

[52] Nithin George, Hyoukjoong Lee, David Novo, Tiark Rompf, Kevin Brown, Arvind Sujeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. Hardware system synthesis from domain-specific languages. In *Proceedings of the 23rd International Conference on Field-Programmable Logic and Applications*, pages 1–8, Munich, September 2014.

[53] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Pearson, forth edition, 2017.

[54] Ramaswamy Govindarajan, Guang R. Gao, and Palash Desai. Minimizing buffer requirements under rate-optimal schedule in regular dataflow networks. *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology*, 31(3):207–29, July 2002.

[55] Graphviz graph visualization software. *https://www.graphviz.org/*.

[56] Mark R. Greenstreet and Kenneth Steiglitz. Bubbles can make self-timed pipelines fast. *Journal of VLSI Signal Processing*, 2(3):139–48, November 1990.

[57] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. Polly - polyhedral optimization in LLVM. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques*, pages 1–6, Chamonix, April 2011.

[58] Tobias Christian Grosser. *Enabling polyhedral optimizations in LLVM*. PhD thesis, University of Passau, 2011.

[59] Sumit Gupta, Nick Savoiu, Nikil Dutt, Rajesh Gupta, and Alex Nicolau. Conditional speculation and its effects on performance and area for high-level synthesis. In *Proceedings of the 14th International Symposium on Systems Synthesis*, pages 171–76, October 2001.

[60] Sumit Gupta, Nick Savoiu, Sunwoo Kim, Nikil Dutt, Rajesh Gupta, and Alex Nicolau. Speculation techniques for high level synthesis of control intensive designs. In *Proceedings of the 38th Design Automation Conference*, pages 269–72, June 2001.

[61] Tae Jun Ham, Juan L. Aragón, and Margaret Martonosi. Decoupling data supply from computation for latency-tolerant communication in heterogeneous architectures. *ACM Transactions on Architecture and Code Optimization*, 14(2):1–27, June 2017.

[62] John Hansen and Montek Singh. Multi-token resource sharing for pipelined asynchronous systems. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 1191–96, Dresden, March 2012.

[63] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, and Hiroaki Takada. Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis. *Journal of Information Processing*, 17:242–54, October 2009.

[64] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, fifth edition, 2011.

[65] Greg Hoover and Forrest Brewer. Synthesizing synchronous elastic flow networks. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 306–11, Munich, March 2008.

[66] Jing Huang, Yuanjie Huang, Yunji Chen, Paolo Ienne, Olivier Temam, and Chengyong Wu. A low-cost memory interface for high-throughput accelerators. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 1–10, New Delhi, October 2014.

[67] Yuanjie Huang, Paolo Ienne, Olivier Temam, Yunji Chen, and Chengyong Wu. Elastic CGRAs. In *Proceedings of the 21st ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 171–80, Monterey, Calif., February 2013.

[68] Donald B Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, March 1975.

[69] Lana Josipović, Atri Bhattacharyya, Andrea Guerrieri, and Paolo Ienne. Shrink it or shed it! Minimize the use of LSQs in dataflow designs. In *Proceedings of the IEEE International Conference on Field Programmable Technology*, pages 197–205, Tianjin, December 2019.

[70] Lana Josipović, Philip Brisk, and Paolo Ienne. From C to elastic circuits. In *Proceedings of the 51st Annual Asilomar Conference on Signals, Systems, and Computers*, pages 121–25, Pacific Grove, Calif., November 2017.

[71] Lana Josipović, Philip Brisk, and Paolo Ienne. An out-of-order load-store queue for spatial computing. *ACM Transactions on Embedded Computing Systems*, 16(5s):125:1–125:19, September 2017.

[72] Lana Josipović, Nithin George, and Paolo Ienne. Enriching C-based high-level synthesis with parallel pattern templates. In *Proceedings of the 26th IEEE International Conference on Field Programmable Technology*, pages 177–80, Xian, December 2016.

[73] Lana Josipović, Radhika Ghosal, and Paolo Ienne. Dynamically scheduled high-level synthesis. In *Proceedings of the 26th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 127–36, Monterey, Calif., February 2018.

[74] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. Speculative dataflow circuits. In *Proceedings of the 27th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 162–71, Seaside, Calif., February 2019.

[75] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. Dynamatic: From C/C++ to dynamically scheduled circuits. In *Proceedings of the 28th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 1–10, Seaside, Calif., February 2020.

[76] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. Synthesizing general-purpose code into dynamically scheduled circuits. *IEEE Circuits and Systems Magazine*, 2021. To appear.

[77] Lana Josipović, Shabnam Sheikhha, Andrea Guerrieri, Paolo Ienne, and Jordi Cortadella. Buffer placement and sizing for high-performance dataflow circuits. In *Proceedings of the 28th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 186–96, Seaside, Calif., February 2020.

[78] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12, Toronto, June 2017.

[79] Timothy Kam, Michael Kishinevsky, Jordi Cortadella, and Marc Galceran-Oms. Correct-by-construction microarchitectural pipelining. *Proceedings of the 27th International Conference on Computer-Aided Design*, pages 434–41, November 2008.

[80] Ryan Kastner, Janarbek Matai, and Stephen Neuendorffer. Parallel programming for FPGAs. *ArXiv e-prints*, arXiv:1805.03648, May 2018.

[81] David C. Ku and G. De Mitcheli. Relative scheduling under timing constraints: Algorithms for high-level synthesis of digital circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(6):696–718, June 1992.

[82] Monica S. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the 1988 ACM Conference on Programming Language Design and Implementation*, pages 318–28, Atlanta, Ga., June 1988.

[83] Vianney Lapotre, Philippe Coussy, Cyrille Chavet, Hugues Wouafo, and Robin Danilo. Dynamic branch prediction for high-level synthesis. In *Proceedings of the 23rd International Conference on Field-Programmable Logic and Applications*, pages 1–6, Porto, September 2013.

[84] Charles E. Leiserson and James B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1-6):5–35, June 1991.

[85] Junyi Liu, Samuel Bayliss, and George A. Constantinides. Offline synthesis of online dependence testing: Parametric loop pipelining for HLS. In *Proceedings of the 23rd IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 159–62, Vancouver, May 2015.

[86] The LLVM Compiler Infrastructure. *http://www.llvm.org*, 2018.

[87] Rajit Manohar and Alain J. Martin. Slack elasticity in concurrent computing. In *Proceedings of the 4th International Conference on the Mathematics of Program Construction*, pages 272–85, London, June 1998.

[88] Mentor Graphics. ModelSim, 2016.

[89] Multi-Level IR Compiler Framework. *https://mlir.llvm.org/*, 2020.

[90] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–80, April 1989.

[91] Mehrdad Najibi and Peter A Beerel. Slack matching mode-based asynchronous circuits for average-case performance. In *Proceedings of the 32nd International Conference on Computer-Aided Design*, pages 219–25, San Jose, Calif., November 2013.

[92] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, Jason Anderson, and Koen Bertels. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, October 2016.

[93] Sune Fallgaard Nielsen, Jens Sparsø, Jonas Braband Jensen, and Johan Sebastian Rosenkilde Nielsen. A behavioral synthesis frontend to the Haste/TiDE design flow. In *Proceedings of the 15th International Symposium on Asynchronous Circuits and Systems*, pages 185–94, Chapel Hill, N.C., May 2009.

[94] Sune Fallgaard Nielsen, Jens Sparsø, and Jan Madsen. Behavioral synthesis of asynchronous circuits using syntax directed translation as backend. *IEEE Transactions on Very Large Scale Integration Systems*, 17(2):248–61, February 2009.

[95] Eriko Nurvitadhi, James C. Hoe, Timothy Kam, and Shih-Lien L. Lu. Automatic pipelining from transactional datapath specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(3):441–54, March 2011.

[96] Il Park, Chong Liang Ooi, and T.N. Vijaykumar. Reducing design complexity of the load/-store queue. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 411–22, San Diego, Calif., December 2003.

[97] Miquel Pericàs, Adrián Cristal, Francisco J. Cazorla, Rubén González, Alexander V. Veidenbaum, Daniel A. Jiménez, and Mateo Valero. A two-level load/store queue based on execution locality. In *Proceedings of the 35th International Symposium on Computer Architecture*, pages 25–36, Beijing, June 2008.

[98] Louis-Noël Pouchet. *Polybench: The polyhedral benchmark suite*, 2012.

[99] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, third edition, 2007.

[100] Andrew R. Putnam, Dave Bennett, Eric Dellinger, Jeff Mason, and Prasanna Sundararajan. CHiMPS: A high-level compilation flow for hybrid CPU-FPGA architectures. In *Proceedings of the 16th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 173–78, Monterey, Calif., February 2017.

[101] C. V. Ramamoorthy and Gary S. Ho. Performance evaluation of asynchronous concurrent systems using Petri nets. *IEEE Transactions on Software Engineering*, 6(5):440–49, September 1980.

[102] C. Ramchandani. Analysis of asynchronous concurrent systems by timed Petri nets. Technical Report Project MAC Technical Report 120, Massachusetts Institute of Technology, February 1974.

[103] B. Ramakrishna Rau. Iterative modulo scheduling. *International Journal of Parallel Programming*, 24(1):3–64, February 1996.

[104] Simha Sethumadhavan, Franziska Roesner, Joel S. Emer, Doug Burger, and Stephen W. Keckler. Late-binding: Enabling unordered load-store queues. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 347–57, San Diego, Calif., June 2007.

[105] John Shalf. The future of computing beyond Moore's law. *Philosophical Transactions of the Royal Society A*, 378(2166):20190061, January 2020.

[106] Jens Sparsø. Current trends in high-level synthesis of asynchronous circuits. In *Proceedings of the 16th IEEE International Conference on Electronics, Circuits, and Systems*, pages 347–50, Yasmine Hammamet, December 2009.

[107] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations. In *International Symposium on High Performance Computer Architecture*, pages 689–702, San Diego, Calif., February 2020.

[108] Mingxing Tan, Bin Liu, Steve Dai, and Zhiru Zhang. Multithreaded pipeline synthesis for data-parallel kernels. In *Proceedings of the International Conference on Computer-Aided Design*, pages 718–25, San Jose, Ca., November 2014.

[109] Mingxing Tan, Gai Liu, Ritchie Zhao, Steve Dai, and Zhiru Zhang. ElasticFlow: A complexity-effective approach for pipelining irregular loop nests. In *Proceedings of the 34th International Conference on Computer-Aided Design*, pages 78–85, Austin, Tex., November 2015.

[110] Linda Torczon and Keith Cooper. *Engineering a Compiler*. Morgan Kaufmann, second edition, 2011.

[111] Richard Townsend, Martha A. Kim, and Stephen A. Edwards. From functional programs to pipelined dataflow circuits. In *Proceedings of the 26th International Conference on Compiler Construction*, pages 76–86, Austin, TX, February 2017.

[112] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 191–202, Philadelphia, PA, May 1996.

[113] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, Santa Margherita Ligure, May 1995.

[114] Arthur H. Veen. Dataflow machine architecture. *ACM Computing Surveys*, 18(4):365–96, December 1986.

[115] Girish Venkataramani and Seth C. Goldstein. Leveraging protocol knowledge in slack matching. In *Proceedings of the 25th International Conference on Computer-Aided Design*, pages 724–29, San Jose, Calif., November 2006.

[116] Muralidaran Vijayaraghavan and Arvind. Bounded dataflow networks and latency-insensitive circuits. In *Proceedings of the 9th International Conference on Formal Methods and Models for Codesign*, pages 171–80, Cambridge, MA, July 2009.

[117] Henry Wong, Vaughn Betz, and Jonathan Rose. Efficient methods for out-of-order load/-store execution for high-performance soft processors. In *Proceedings of the IEEE International Conference on Field Programmable Technology*, pages 442–45, Kyoto, December 2013.

[118] Xilinx Inc. *Vivado Design Suite User Guide: High-Level Synthesis*, 2018.

[119] Xilinx Inc. *Vivado High-Level Synthesis*, 2018.

[120] Zhiru Zhang and Bin Liu. SDC-based modulo scheduling for pipeline synthesis. In *Proceedings of the 32nd International Conference on Computer-Aided Design*, pages 211–18, San Jose, Calif., November 2013.

Lana Josipović

lana.josipovic@epfl.ch

Education

2015 - 2020 **École Polytechnique Fédérale de Lausanne, School of Computer and Communication Sciences**
PhD in Computer and Communication Sciences.

2014 - 2015 **Technische Universität München, Department of Electrical, Electronic and Computer Engineering**
Master of Science in Electrical Engineering and Information Technology (Erasmus+ exchange program).

2013 - 2015 **University of Zagreb, Faculty of Electrical Engineering and Computing**
Master of Science in Electrical Engineering and Information Technology.

2010 - 2013 **University of Zagreb, Faculty of Electrical Engineering and Computing**
Bachelor of Science in Electrical Engineering and Information Technology.

Awards and Honors

2020 **Best Paper Award** at the 28th Intl. Symposium on Field Programmable Gate Arrays.

2018 **Google PhD Fellowship for Systems and Networking** for exceptional PhD research in Computer Science.

2018 **Best Paper Award Nominee** at the 26th Intl. Symposium on Field Programmable Gate Arrays.

2017 **Best Paper Award Nominee** at the Intl. Conf. on Compilers, Architectures, and Synthesis for Embedded Systems.

2015 **EDIC doctoral fellowship,** IC School, EPFL, for new PhD students with exceptional academic record.

2015 **Google Anita Borg Memorial (Women Techmakers) Scholarship,** for excellent academic achievements, leadership abilities, and community engagement.

2010 - 2015 **Dean's awards,** Univ. of Zagreb, for outstanding achievements in the BSc ('11, '12, '13) and MSc ('15) studies.

2013 - 2015 **Scholarship for gifted students,** Univ. of Zagreb, for the **top 1%** most successful students ('14, '15).

Experience

2015 - 2020 **Researcher/PhD Candidate at Processor Architecture Laboratory, EPFL**. Developed new high-level synthesis (HLS) techniques to create efficient accelerators from high-level programming languages. Built an HLS tool which generates dynamically scheduled dataflow circuits out of C/C++ code and overcomes some of the limitations of existing HLS solutions. My HLS compiler and all related documentation are available at http://dynamatic.epfl.ch/.

2019 **Research Intern at Xilinx, San Jose, CA.** Built an MLIR-based compiler for translating high-level languages into hardware designs which target dataflow-oriented programmable chips. I developed a transformation of a standard compiler intermediate representation into a dataflow dialect which is based on my PhD work; it is now available as part of a new hardware compiler suite at https://github.com/llvm/circt.

2018 **Research Intern at Microsoft Research, Redmond, WA**. Performed research associated with project Catapult, focused on tools for creating FPGA solutions and deploying them at scale in Microsoft's cloud servers.

2014 - 2015 **Electrical Engineer at Max Planck Institute for Biochemistry, Munich**. Developed technical solutions for mass spectrometry-based proteomics, applied for analysis of protein structures that cause cancer and diabetes.

2014 **Intern at Processor Architecture Laboratory, EPFL.** Developed a tool for automatic transistor sizing for commercial and non-standard FPGA architectures.

Languages

Croatian (native), **English** (proficient), **German** (proficient), **French** (beginner), **Spanish** (beginner).

Publications

2020 **Lana Josipović**, Andrea Guerrieri, and Paolo Ienne. Synthesizing general-purpose code into dynamically scheduled circuits. *IEEE Circuits and Systems Magazine (CASM).* To appear.

2020 **Lana Josipović**, Shabnam Sheikhha, Andrea Guerrieri, Paolo Ienne, and Jordi Cortadella. Buffer placement and sizing for high-performance dataflow circuits. In *Proceedings of the 28th ACM/SIGDA Intl. Symposium on Field Programmable Gate Arrays (FPGA'20)*, pages 186–96, Seaside, Calif., February 2020. **Best Paper Award.**

2020 Jianyi Cheng, **Lana Josipović**, George A. Constantinides, Paolo Ienne, and John Wickerson. Combining dynamic & static scheduling in high-level synthesis. In *Proceedings of the 28th ACM/SIGDA Intl. Symposium on Field Programmable Gate Arrays (FPGA'20)*, pages 288–98, Seaside, Calif., February 2020.

2020 **Lana Josipović**, Andrea Guerrieri, and Paolo Ienne. Dynamatic: From C/C++ to dynamically scheduled circuits. In *Proceedings of the 28th ACM/SIGDA Intl. Symposium on Field Programmable Gate Arrays (FPGA'20)*, pages 1–10, Seaside, Calif., February 2020.

2019 **Lana Josipović**, Atri Bhattacharyya, Andrea Guerrieri, and Paolo Ienne. Shrink it or shed it! Minimize the use of LSQs in dataflow designs. In *Proceedings of the IEEE Intl. Conference on Field Programmable Technology (FPT'19)*, pages 197–205, Tianjin, China, December 2019.

2019 **Lana Josipović**, Andrea Guerrieri, Paolo Ienne, and Jordi Cortadella. Performance optimization of dataflow circuits. In *Proceedings of the Intl. Workshop on Logic Synthesis (IWLS'19)*, pages 146–53, Lausanne, June 2019.

2019 **Lana Josipović**, Andrea Guerrieri, and Paolo Ienne. Speculative dataflow circuits. In *Proceedings of the 27th ACM/SIGDA Intl. Symposium on Field Programmable Gate Arrays (FPGA'19)*, pages 162–71, Monterey, Calif., February 2019.

2018 **Lana Josipović**, Radhika Ghosal, and Paolo Ienne. Dynamically scheduled high-level synthesis. In *Proceedings of the 26th ACM/SIGDA Intl. Symposium on Field Programmable Gate Arrays (FPGA'18)*, pages 127-36, Monterey, Calif., February 2018. **Best Paper Award Nominee.**

2017 **Lana Josipović**, Philip Brisk, and Paolo Ienne. From C to elastic circuits. In *Proceedings of the 51st Annual Asilomar Conference on Signals, Systems, and Computers*, pages 121-25, Pacific Grove, Calif., October 2017.

2017 **Lana Josipović**, Philip Brisk, and Paolo Ienne. An out-of-order load-store queue for spatial computing. In *Proceedings of the Intl. Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES'17)*, Seoul, Korea, October 2017. See ACM TECS paper below. **Best Paper Award Nominee.**

2017 **Lana Josipović**, Philip Brisk, and Paolo Ienne. An out-of-order load-store queue for spatial computing. *ACM Transactions on Embedded Computing Systems (TECS'17)*, 16(5s):125:1–125:19, September 2017.

2017 **Lana Josipović**, Philip Brisk, and Paolo Ienne. An out-of-order load-store queue for spatial computing. In *Proceedings of the 25th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'17)*, page 134, Napa, Calif., April 2017.

2016 **Lana Josipović**, Nithin George, and Paolo Ienne. Enriching C-based high-level synthesis with parallel pattern templates. In *Proceedings of the 26th IEEE Intl. Conference on Field Programmable Technology (FPT'16)*, pages 177–80, Xi'an, China, December 2016.