POLITECNICO DI TORINO

Master Degree course in Computer Engineering

Master Degree Thesis

From Gated-SSA to Out-of-Order Dataflow Circuits

Supervisors
Prof. Mariagrazia Graziano
Prof. Paolo Ienne
Ayatallah Elakhras

Candidate
Giacomo SANSONE

Academic Year 2024-2025

Abstract

High Level Synthesis (HLS) is the process of generating hardware from a programming language. This can help during the circuit's prototyping phase or allow engineers who are not specialized in hardware to develop circuits that will run on FPGAs. In the past few years, a new trend in the HLS world arose, allowing hardware components to communicate through a handshake. This approach, called *dynamic HLS*, removes the need of a controller in the circuit, allowing for better runtime (up to 5$\times$ faster) at the expense of increased area utilization ($2\times \sim 10\times$).

This thesis starts from this context, focusing on *Dynamatic*, an open-source *dynamic* HLS tool developed at the Processor Architecture Laboratory in EPFL.

Recently, a new synthesis strategy has been developed which aims to reduce the resulting circuit's complexity by interconnecting hardware components according to their *producer-consumer* relationships. The implementation of these algorithms is not available in Dynamatic, thus the first goal of this thesis is to fill this gap. The result provides an average 0.7$\times$ the execution time of the baseline and 0.87$\times$ area utilization.

In 2024, the problem of *out-of-order* execution was addressed in the context of Dynamatic. The result was a framework which allows execution to go out-of-order in specific areas of the circuit, while maintaining functional correctness through tagging and aligning. However, such work does not provide a way to determine when it is worth going out-of-order, or when timing improvements are possible. This thesis also aims at filling this research gap. Through analytical study of the circuit structure, it is possible to obtain a static analysis of the expected improvement when applying the *out-of-order* execution to the problem of loop pipelining. The algorithm which has been developed shows an average error of 5% when compared to the simulated circuit, whose values are used as a reference.

Contents

Chapter 1

Introduction

1.1 Background

High-level synthesis (HLS) [1] allows one to generate hardware from a programming language, such as C, C++, or Python. For instance, having a C function that computes a multiplication between two matrices stored in memory, it is possible to translate it to a circuit - defined using a Hardware Description Language - that implements the same functionalities. The ease of obtaining hardware with this approach fits well with FPGAs, reconfigurable hardware that allows fast development and time-to-market compared to the full ASIC process.

This kind of compiler has two main benefits:

1. It allows people who are into software development but not into hardware to obtain an FPGA accelerator without a deep understanding of the underlying system. In this way, FPGAs can be adopted by a wider audience, providing better execution time than software while being cheaper than ASIC.

2. FPGA prototyping goes way faster with an HLS tool. A hardware engineer has to ensure that the algorithm is fully working before moving it into a piece of hardware: by using such a compiler, a rough prototype of the system can be obtained, and then further optimized using the engineer's knowledge and insights.

In general, HLS cannot replace a hardware engineer, but it is a valuable tool in the design flow. The current market has many examples of such tools. Vitis™ HLS [2] by AMD is probably the most known example, due to its integration with the Vivado design suite. Other examples of such tools are Stratus™ HLS [3] by Cadence and Catapult HLS [4] by Siemens. The trivial way of implementing a circuit out of a sequential piece of code consists of the following flow: operations are translated into circuits components, where components corresponding to data-dependent operations are interconnected, forming a datapath; following this, a controller is needed to determine the schedule of execution of components, honoring the program's control flow decisions.

Such a high-level view of the flow cannot address all the issues that might arise; however, it depicts the fact that the resulting circuit is made of a *datapath* and a *controller*. The controller is said to be *static* whenever the timing of each operation is scheduled at compile time. This is a conservative approach since it does not take into account the possible alternatives in the flow that only arise at runtime. For instance, let's suppose that a multiplication is executed only if a value read by the memory is 0. As the compiler cannot know the value itself, it is forced to allocate some clock cycles for the arithmetic operation; if this is the case, those clock cycles are lost.

Although such naïve implementation can be improved, it always results in a more complicated controller, difficult to debug and understand. A recent HLS approach replaced the centralized controller with dynamically scheduled circuits, and has proven effective in irregular and control-dominated applications over its statically scheduled counterpart [5].

The idea behind such HLS is to implement an *elastic/dataflow* circuit made of synchronous components that interact with each other through a handshake. An operation can run as soon as its operands are *valid*, without relying on a static timing imposed by a controller; an operation can provide a result as soon as the successor is *ready* to receive. Components exchange *tokens* this way. Such a system leads to major benefits in terms of execution time and loop initialization intervals (up to 5×), while requiring more LUTs and FFs ($2\times \sim 10\times$) with respect to a static approach [5].

This methodology revives the idea of dataflow circuits (also known as *elastic* or *latency-insensitive* circuits), which implement a distributed control mechanism through handshake signals [6].

Since the initial publications on the topic, many authors worked on such a compiler, implementing analysis and functionalities which further reduced both the required area and the execution time of the final circuit [7–14]. This effort led to *Dynamatic*, an open-source C-to-dataflow circuits compiler [15, 16]. Such a compiler was first developed and maintained in LLVM. Most of the works cited so far are based on this LLVM version. Recently, MLIR (Section 2.3) has been adopted as the framework for the compiler.

This thesis starts from such a context to investigate new possibilities to improve the resulting circuit of Dynamatic. In particular, a recent proposal in the dynamic HLS community defined a new dataflow circuit generation strategy that delivers data *directly* between the data-dependent components, skipping irrelevant control decisions [10, 12]. It achieves tangible improvements in execution time, by an average value of 27% [10].

1.2 Problem Statement

Dynamatic has been open-source since its first release, in its LLVM version [5]. The switch to MLIR [17] was a logical step for the growth of the compiler in the open-source world: this framework allows for better maintainability together with ease in development (Section 2.3). Yet, the MLIR-based tool is still in its early stage, and it misses many of the advances done in research including efficient dataflow circuit generation strategies [10, 12] and out-of-order execution [14].

Although there has been a lot of research advances towards improving the performance of dataflow circuits by adding support for speculation and multithreading [8, 10, 14, 18, 19], at the cost of increased overhead, little is known about analysis to identify when such techniques are useful. Specifically, recent research proposed analysis to determine when speculation is useful in HLS [20]; yet, the question of where and when out-of-order execution is useful in HLS remains open.

1.3 Goals

The goal of the thesis is thus double: on the one hand, it aims to provide a clear and reliable implementation of *Fast Token Delivery* [12] and *Straight to the Queue* [10] in the MLIR-based Dynamatic; on the other hand, on top of these new features, it wants to provide a way to determine if it is worth introducing out-of-order execution in a circuit, considering the trade-off between the area overhead and the timing improvement.

1.4 Implementation and Research Methodology

The implementation side is mostly dependent on the daily updates of the main compiler's codebase. The project is handled by 2 research groups at EPFL and ETHZ, involving more than 20 full-time contributors. All the intended additions must be discussed and approved by the underlying community, in order not to compromise negatively someone else's work.

The implementation will provide numerical results that can be used to compare the features with the stable version of the compiler. The results should be reasonably comparable to the ones presented in the original papers - with some minor changes due to implementation details and the fast development of the tool in the years since the publications.

Once the implementation is completed, the research part starts. This consists in extracting relevant features from the circuit which can point out regions of the circuit where to apply out-of-order techniques from [14]. This should consist of a compile-time static analysis, relying both on the abstract behavior of the components in the circuit and the context in which the circuit is planned to be inserted.

All the results will be open-source and reproducibile.

1.5 Delimitations

The implementations of *Fast Token Delivery* [10] and *Straight to the Queue* [12] do not consider the full integration with the rest of Dynamatic flow. Some of the other passes should be modified and adapted to work with the circuits obtained through these methods. Due to the size of the codebase, this task is not feasible during the time of the thesis.

In the research part, an algorithm will be developed to perform static analysis. This methodology proves the feasibility of the project, but it is still premature and it does not allow for the analysis of any dataflow circuit. In a later stage, the work should be integrated with a structured mathematical framework for network analysis, such as *Network Calculus* [21].

1.6 Structure of the Work

The thesis is organized as follows:

- Chapter 2 provides a background to the thesis, referencing all the studies that needed to be done before starting the project. In particular, it starts with a discussion on compilers, data structures, and tools that are fundamental to later discussions; then it provides an overview of Dynamatic.
- Chapter 3 is about the implementation of *Fast Token Delivery* [10], considering the algorithm first and some technical details later.
- Chapter 4 is about the implementation of *Straight to the Queue* [12], considering the algorithm first and some technical details later.
- Chapter 5 compares the implemented work with the simple dataflow circuit generation strategy in Dynamatic, giving insights into the numerical results of the analysis.
- Chapter 6 is an introduction to the problem of Thread-Level Parallelism in dataflow circuits, which should be solved using the aforementioned out-of-order techniques.
- Chapter 7 provides the research contribution of this thesis, a technique that statically analyzes the circuit to provide insights about the applicability of [14]. This comes together with an experimental validation of the proposed technique.
- Chapter 8 summarizes the scope and results of this thesis.

Chapter 2

Background

This chapter is organized as follows. In Section 2.1, an overview of compilers is provided; readers with a background in electronics engineering might not know about the topic, since this is considered a pure computer science matter. In Section 2.2, the gated-static single assignment (GSA) is explained, being the main theoretical technique adopted for the thesis. In Section 2.3, an introduction to MLIR is provided, being the framework adopted to build Dyanamtic. In Section 2.4, an overview of Dyanamtic is provided, highlighting the main steps in the process of converting a piece of software into hardware using the *dynamic HLS* methodology.

2.1 Compilers Background

This section is inspired by [22], which has been the main compiler reference for this thesis.

2.1.1 Compiler Structure

A compiler is a piece of software to translate a source program written in language X into a program written in Y, while maintaining the same functionalities. The term *transpiler* is used when Y is another high-level programming language (for instance, to move from Typescript to JavaScript); a compiler usually targets the ISA of a processor, so that, having Assembly as the target output, the initial program can be executed on a CPU.

Since a compiler is the central tool of any software development process, it is fundamental to guarantee its correctness; programmers rely on such a tool to execute their software, and they take for granted that an issue at run-time is only due to their faults, not on the underlying system. This is expressed through the following principle: «a compiler must preserve the meaning of the program being compiled» [22]. If such a rule is not fulfilled, paradoxical consequences can be reached. For instance, inserting a bug in the translation is equivalent to translating the whole program into a `nop`, which is a conversion that does not require a compiler in the first place.

It is reasonable to divide a compiler into three main successive elements: the frontend, the optimizer/middle end, and the backend (Figure 2.1). This distinction comes from a recent belief according to which a compiler should be designed using many single-scoped passes, with each of them targeting a single problem at one time.

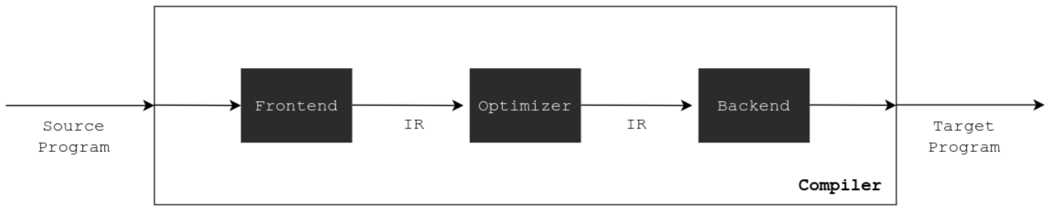


Figure 2.1: Representation of the structure of a compiler, inspired from [22].

Frontend

The first step for a compiler is to understand the syntax of the source, trying to figure out as many faults on the programmer's side as possible (sometimes also about the semantics). In this context, the compiler is fully aware of the rules - grammar - to say when a source program is *well-formed*. The more accurate this part of the compiler is, the easier the job of the programmer: by providing some useful insights into the errors found in the source program, it will be faster to fix them. On the contrary, by only stating that *there is an error somewhere* the programmer has no clue where to start debugging.

The *lexer* or *tokenizer* is in charge of distinguishing the different tokens as they are sequentially listed in the program; the *parser* puts them together, moving them to a form that is interpretable by the following stages. Let's consider this C assignment: `int a = 10 * b - 20 * c;`. The tokenizer provides a list of the tokens found in the source code, following a set of rules for the way tokens can be formed. `int` is recognized as a keyword, `a` `b`, and `c` as names of variables, `10` and `20` as numbers, `=` `-` and `*` as operators; `;` is a delimiter, to inform that the statement is concluded. The parser can use the list of tokens to associate a specific rule. In this case, it finds a sequence made of a variable type, a variable name, an assignment operator, a (well-formed) numerical expression, and a delimiter: this is how variable declarations are described in C grammar. While parsing, a type check and a *context-sensitive analysis* are performed too: not only `b` and `c` must exist before their usage, but their type must be coherent with the expression - you cannot multiply `int` and `int*` according to C's rules.

Once this phase is concluded, it is possible to state the syntactical correctness of the input program. This does not imply semantic correctness either.

Middle end

The frontend keeps the structure of the program as it was delivered by the programmer, without any modification. This is usually suboptimal due to many reasons. The programmer might make mistakes in the first place: redundant code has no impact on functionalities, but it might lead to larger target programs; some loops might be suppressed and substituted with direct expressions; some code might be *dead*, thus worth removing. However, some optimizations might also be related to the target itself, exploiting some characteristics the programmer is not aware of (what if the programmer relied on a routine to compute a multiplication between `a` and `b`, while the target ISA has an *ad hoc* operation?).

The middle end usually targets one optimization at a time, by first analyzing the input and possibly applying the semantics-invariant transformation. Each transformation is applied based on a definition of efficiency which must be improved. This might be about an estimation of the resulting execution time or the resulting code size.

Backend

Once the source code has been parsed and optimized, it can be lowered to the target language. This can be either an ISA of a CPU (which is what is going to be used as a reference) or some HDL in the case of an HLS compiler.

Each operation has to be translated into one or more assembly instructions. This is straightforward when it comes to simple arithmetic operations, but what if the target does not support the operation directly? You might want to divide two numbers on a RISC-V architecture without the *M* extension: in this case, the compiler should provide a sort of sub-routine to functionally have the same output.

Not only do instructions have to be properly selected, but they also have to be scheduled according to the target CPU architecture: an accurate knowledge of the underlying hardware can reduce the execution time by reducing the number of hazards in the pipeline.

While performing the above two phases - instruction selection and scheduling - it is common to work as if the number of registers was infinite. Each CPU has a limited set of general purpose registers in its ISA, and they have to be allocated to each instruction without compromising their values. This can lead to some *fill/spill* operations to the main memory, in case the amount of registers available is not enough to guarantee the execution. Moving the content of registers back and forth in the memory has a huge cost, and it should be limited: register allocation is a fundamental problem when it comes to the resulting execution time.

2.1.2 Intermediate Representations and Data Structures

The source code in the textual format is humanly-readable but hard to be handled by the compiler itself directly. This is why there are many different intermediate representations which are used at different stages of the compilation flow. Next to different IRs that allow simple transformations, there are also a set of data structures fundamental to properly analyze the structure of the source code, and extract information out of it. This section briefly covers some of these.

It is useful to visualize some of these data structures throughout this explanation. The piece of code in Listing 2.1 is used as a reference.

```
int f(int upper_bound) {
    int i = 0, sum = 0;
    while(i < upper_bound) {
        if(i % 2 == 0)
5             sum =  sum + upper_bound * 20;
6         else
7             sum -= upper_bound;
8     }
9     i++;
10    return sum;
11 }
```

Listing 2.1: Reference code for IR explanation.

Abstract Syntax Tree

As a program is parsed and the different grammar rules are matched over the source code structure, the obtained knowledge is maintained in a tree structure called *abstract syntax tree*. This tree contains everything that comes out of the parsing stage, keeping only what is useful in the first place and removing anything else (such as the semicolon delimiter in C).

Let's consider the `if-then-else` statement in Listing 2.1 from lines 4 to 7. The result of the parsing stage is shown as an abstract syntax tree in Figure 2.2. The if statement requires three components: an expression, a `then` statement, and an (optional) `else` statement. Each statement is a variable assignment, thus having a left value on the left side and an expression on the right side. Notice how the tree also incorporates the information on the operators' relationship: since the multiplication has higher priority than the addition, it is at a lower level of the tree. If the tree is traversed in a depth first fashion, this makes sure that the multiplication is handled before the sum, as the associativity rules of the system want.

It is worth noting that this is only a frontend matter, with no implications in the two subsequent compiling stage; since the HLS flow mostly involves these latter two parts, no more details will be given (usually, different compilers share the same frontend and design the rest of the flow; as long as the syntax of the language is the same, it is not worth to reinvent the wheel).

Control Flow Graph

A *Basic Block* (BB) in the source code is a maximal length sequence of branch-free code. The operations in a BB are always executed together because, as the definition implies, no branch impacts the flow. Depending on some conditions in the code, some BBs can be executed or not. For instance, referring to the example code, if the value of `upper_bound` is negative, none of the operations within the while loop are executed. Such a flow of execution among BBs according to some conditions is encoded in the *Control-Flow Graph* (CFG).

A CFG is a directed graph $G = (N, E)$; each node $n \in N$ is a BB; each edge $e = (n_i, n_j) \in E$ is a possible transfer of control from n_i to n_j.

It is reasonable to say that a node in the CFG can have either 0 successor (the last node of the execution), 1 successor (it terminates with an unconditional branch so that n_j is always executed after n_i) or 2 successors (n_j if a condition is true, n_i if it is false).

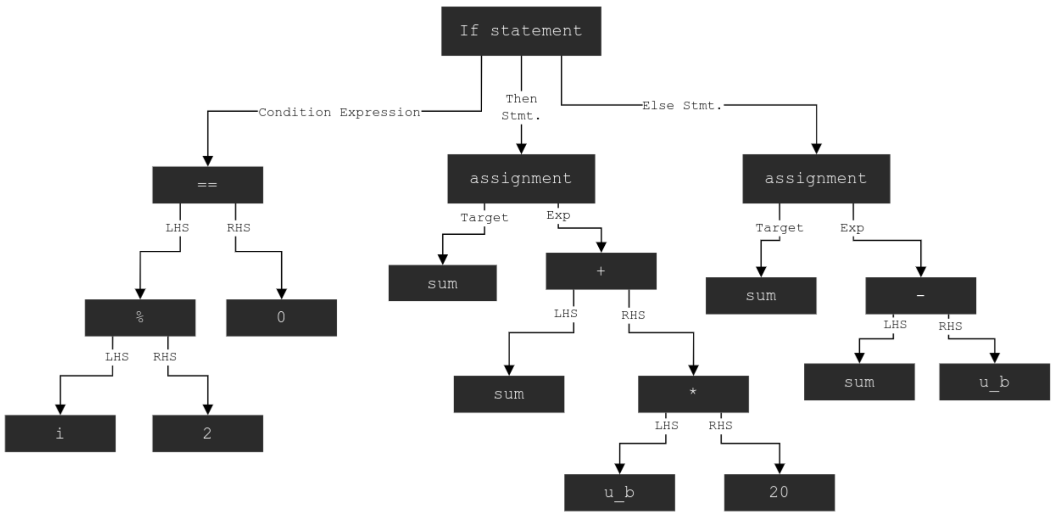


Figure 2.2: This shows an example of an Abstract Syntax Tree parsing an `if-then-else` statement.

With some boolean manipulation, the $N > 2$ successors case can be moved to a cascade of nodes having two successors each.

The CFG of Listing 2.1 is shown in Figure 2.3. Black arrows correspond to unconditional branches, green (red) arrows to branches taken if the corresponding condition is true (false). Block 0 refers to the initialization of the variables; block 1 is in charge of checking the `while` condition, while block 2 is for the `if` condition; block 3 contains the `then` case, block 4 is the `else` case; block 5 is a merging node for the end of the if statement in which `i` is incremented, while block 6 contains the `return` statement.

In this context, each CFG has an *entry node* (without predecessors) and an *exit node* (without successors).

Dominator/Postdominator Tree

Starting from the CFG, many insights on the execution of the code can be obtained. Depending on the specific required analysis, some information might be useful or not.

The domination theory (also summarized in [23]) is often useful. A node v *dominates* another node w if every path from the entry node of the CFG to w contains v. A node u *post-dominates* w if every path from w to the exit node of the CFG contains u. u *properly dominates* w is u dominates w but $u \neq w$; the same goes for a *proper post-domination.*

A node v is an *immediate dominator* of w if v dominates w and every other dominator of w also dominates v; a node u is an *immediate post-dominator* of w is u post-dominates w and every other post-dominator of w also post-dominates u.

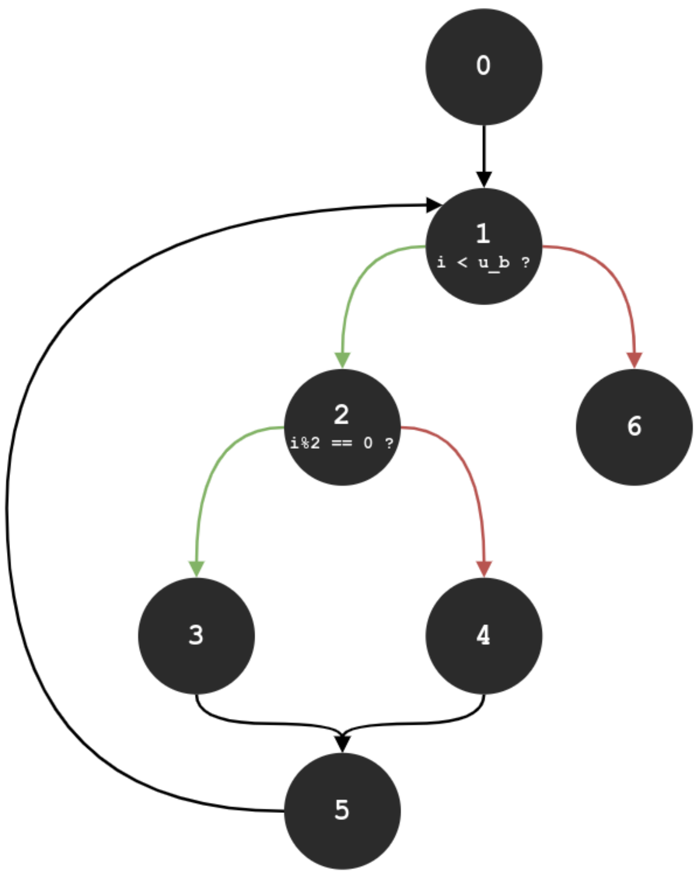


Figure 2.3: This CFG is made of 7 BBs, with 2 conditional branches depending on the corresponding conditions; `BB1` is the entry node, `BB6` is the end node.

Except for the entry node in the CFG, each node has a unique immediate (post)-dominator. This allows to build both a dominator and a post-dominator tree, in which there is an edge between u and v if u is the immediate (post)-dominator of v.

The dominator tree of Listing 2.1 is shown in Figure 2.4, while the post-dominator tree is shown in Figure 2.5.

Some optimizations between BBs can only be performed if certain domination relationships hold. As an example with respect to Figure 2.3: if the same variable is set both in the BBs 1 and 2, it is guaranteed that the value used in the BBs 3, 4, and 5 are the ones set by BB 2 since 2 is an immediate dominator of these nodes; on the contrary, if the same variable is set both in BB 2 and 4, it is not possible to state for sure which value will be used in BB 5, since 4 does not dominate 5.

The concept of domination is also useful to define the control dependency analysis [24]. Given two nodes u and v, v is control dependent on u if the following two conditions hold: there exists a directed path from u to v with any node n in the path (excluding u and v) post-dominated by v; u is not post-dominated by v.

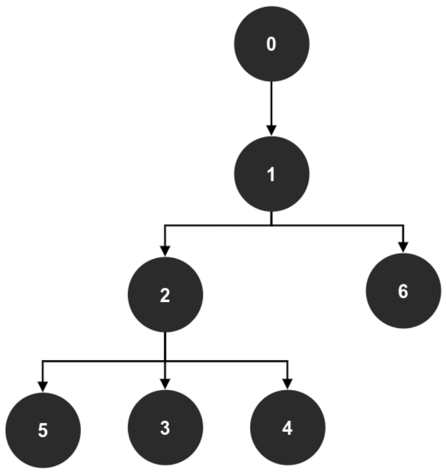


Figure 2.4: The dominatore tree in the picture refers to the example from Figure 2.3.

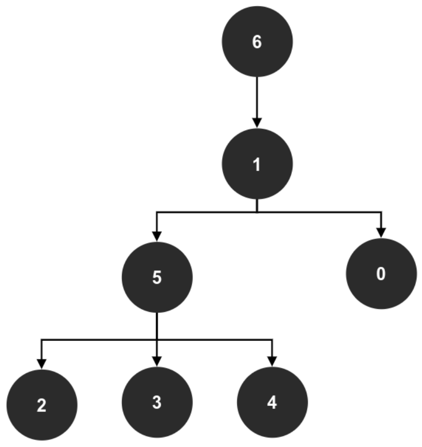


Figure 2.5: The post-dominatore tree in the picture refers to the example from Figure 2.3.

When u is control dependent on v, v has two exits, and u gets executed only by going through one of them. As [24] states, this analysis is fundamental to detect areas of the code that can be run in parallel, fused in the same loop, or moved without affecting final semantics.

Linear IRs

So far only graphical representations of the source program have been considered. However, since both the source and target code are *linear*, it is useful to conceive some standard linear representations too. This is also useful to further manipulate the code during the optimization phases.

Assembly is an example of a linear representation, as there is a clear sequential ordering of the instructions. While having this characteristic, it also allows to represent control flows, through branches and labels. For this reason, it is simple to come up with an algorithm to rebuild the original CFG out of an assembly code (it is a matter of identifying the BBs terminated by a branch and inserting an edge for each possible branch successor).

However, a linear IR can also be expressed at a higher level of abstraction than the well-known assembly. MLIR (Section 2.3) would represent Listing 2.1 as shown in Listing 2.2. The meaning of each operation and its mapping to the original code should be simple to understand. Each operation defines a typed value (starting with `%`); `cf.br` represents an unconditional branch; `cf.cond_br` is a conditional branch.

```
module {
  func.func @f(%arg0: i32 {arg_name = "upper_bound"}) -> i32 {
    %c1 = arith.constant                        1: index
    %c2 = arith.constant                        2: index
    %c0 = arith.constant                        0  index
    %c20_i32 = arith.constant                   20: i32
    %c0_i32 = arith.constant                    0: i32
    cf.br ^bb1(%c0, %c0_i32 : index, i32)
  ^bb1(%2: index, %3: i32):                     // 2 preds: ^bb0, ^bb6
    %0 = arith.index_cast %arg0                 : i32 to index
    %4 = arith.cmpi ult, %2, %0                 : index
    cf.cond_br %4, ^bb2, ^bb6
  ^bb2:                                         // pred: ^bb1
    %5 = arith.remsi %2, %c2                    : index
    %6 = arith.cmpi slt, %5, %c0                : index
    %7 = arith.addi %5, %c2                     : index
    %8 = arith.select %6, %7, %5                : index
    %9 = arith.cmpi eq, %8, %c0                 : index
    cf.cond_br %9, ^bb3, ^bb4
  ^bb3:                                         // pred: ^bb2
    %1 = arith.muli %arg0, %c20_i32             : i32
    %10 = arith.addi %3, %1                     : i32
    cf.br ^bb5(%10 : i32)
  ^bb4:                                         // pred: ^bb2
    %11 = arith.subi %3, %arg0                  : i32
    cf.br ^bb5(%11 : i32)
  ^bb5(%12: i32):                               // 2 preds: ^bb3, ^bb4
    %13 = arith.addi %2, %c1                    : index
    cf.br ^bb1(%13, %12 : index, i32)
  ^bb6:                                         // pred: ^bb1
    return %3                                   : i32
  }
}
```

Listing 2.2: Reference code for linear IR explanation.

The level of abstraction chosen also implies which optimizations can be performed by the compiler. Also, the naming scheme chosen for the representation can show or hide some optimization opportunities. The most adopted way of naming values in a linear intermediate representation is called *static single assignment* (SSA).

Static Single Assignment

As the name suggests, within SSA [25] each value can be assigned only once and used multiple times. As a consequence, when a name is used, it is possible to understand uniquely where the value was set.

However, these requirements clash with the usual programming convention for which the value of a variable changes over time. Concerning Listing 2.1, the variable `i` is first set and then incremented multiple times, thus it represents different values at different instants of time. This is a consequence of control flow: which is the correct value to be used for `i`?

This problem is solved by adding new elements in the linear code, the ϕ functions, which *merge* into a single definition values defined in multiple places. A wide discussion on ϕ functions can be found in [26]. Through a ϕ function or gates, it is possible to merge the value of the loop-variable `i` from its initialization value and its increment. The same holds for the variable `sum` which is updated in two different BBs. At this level of abstraction, ϕ works as an oracle, always providing the correct value to be used for the usage of that variable.

ϕ functions can be both *implicit* and *explicit*. Listing 2.2 shows an example of implicit ϕ functions, in which different values of the same variable are merged through some block arguments. When the control flow moves from one block to another, the values of the arguments for the ϕ nodes of the destination block must be provided. In that example, `%2` is the name for `i`, while `%3` is the name for `sum`. Explicit ϕ functions automatically gather the correct value each time they are executed. For instance, instead of the block argument `%2` there would be: `phi(%c0, %13)`. Each time `BB1` runs, it automatically understands which is the correct value to be used, thanks to the edge which led to it.

2.2 Gated Static Single Assignment

2.2.1 The Idea

In Section 2.1.2 it was stated both the importance of the SSA representation and its oddities when it comes to implementation. In many contexts, there are no limitations to such an approach, and a wide literature [27] is available on how to exploit SSA to build a compiler's middle end.

However, it should be clear from a computer science perspective that encoding information in different ways always leads to different functionalities and results, each of them with a different level of efficiency. For instance, SSA is not the best existing way to implement a fast interpreter out of a linear code. Such a system needs to know precisely which definition of a variable is to be used, and tracking down the preceding BB starting from the CFG can be time consuming.

Ottenstein et. al. [28] invented an alternative form to SSA, which was called *gated static single assignment* (GSA). Rather than having only one merging function, ϕ, the authors defined multiple of them, each with a specific deterministic meaning.

- μ is a gate with three arguments: a predicate P, an initialization value v_0, and an update value v_{iter}. This gate is used for loop headers, to determine which value is to be used at each loop iteration. The predicate P is true when the loop is to be executed - thus a batch of iterations is launched; for the first iteration of the loop, the value v_0 is used; in all the other iterations until P gets false, v_{iter} is used. In this way, each time a loop is run, the value to be used can be deterministically provided.

- γ is a gate with three arguments: a predicate P, a *true* value v_{true}, and a *false* value v_{false}. The result of the gamma function gets v_{true} when P is evaluated as *true*, v_{false} otherwise. This gate is used to merge values coming from `if-then-else` statements.

Not only the conditions to pick one of the available values are now made explicit, thus leading to a clearer view of the data flow (together with a full independence from the control-flow decisions); but also having some explicit boolean conditions might result in more opportunities for later optimizations. While GSA has been used for much research since its invention, the availability of more advanced techniques based on SSA led to its underestimation. However, some recent results in the context of HLS compilers [10, 19] show that its features are still to be fully exploited.

2.2.2 How to Implement GSA

The aforementioned GSA definition does not automatically explain how to obtain the gate functions together with their relative predicates. In particular, two alternative methods can be adopted: either the gates are constructed out of the CFG only, without relying on previous dataflow analysis [23], or the SSA version is built first, and then the ϕ functions are translated either into μ or γ functions [29, 30].

Since this thesis is based on GSA, a custom MLIR analysis pass was built to obtain such an analysis. The starting point of the pass is the *unstructured control flow* dialect, with its implicit SSA encoding. As the information about ϕ placement is already available, the second approach was adopted, with a custom algorithm that will be explained in Section 3.3.

2.2.3 A Recent Comeback

Recently, many research papers on the topic of GSA were published. In particular:

- In [31], the authors highlight the fact that many different definitions of the GSA gates were provided in different papers, without a clear agreement. For this reason, formal semantics for GSA is provided and mechanised in Coq, together with a way to translate SSA into GSA.

- In [19], GSA is the starting point to implement speculation in circuits resulting from an HLS compiler. Let's suppose that two alternative execution branches can run because of the availability of data, while the control information is still to be produced. A form of speculation can be implemented by starting the execution of the branches (whose results will be then merged through a γ function) and then flushing one of the two as the predicate for the final gate was produced. While being similar to other speculative approaches (such as [8]) the explicit usage of GSA makes the analysis immediate, as the predicate information to state the correctness of the speculation is already available.

2.3 MLIR

Dynamatic was first developed in the context of LLVM and then moved to MLIR. Since this is the framework of the thesis from a development perspective, it is worth giving a general overview of it. The information provided in this section refers to the original MLIR publication [17].

2.3.1 The Purpose of the Framework

LLVM is one of the main infrastructure adopted to build compilers. While providing an enormous set of classes, functions, and operators to implement the result, one of its main ideas is to rely on one single intermediate representation, the *LLVM IR*. This is an abstraction level that aims at fulfilling all the different purposes a compiler design might have.

However, as was mentioned in Section 2.1, different problems require different models, either at a higher or lower level of abstraction. This is why many programming languages require their own IR, without relying directly on the LLVM's one. The addition of an IR requires an integration with the existing infrastructure (a conversion mechanism), which might be an excessive overhead.

MLIR aims to «make it cheap to define and introduce new abstraction levels, and provide *in the box* infrastructure to solve common compiler engineering problems» [17]. You can define an IR by introducing a new dialect, and integrating it with all the available operations; all the IRs are also SSA-based.

According to the authors, the following principles were followed while defining MLIR:

- *Parsimony.* The number of available components is limited, making most of the environment customizable according to different needs. This makes the infrastructure great both for software compilers and HLS compilers. For the same reasons, to be as versatile as possible, canonicalization is left to the IR definition, so that no rules have to be followed *a priori*.

- *Progressivity.* Each structure and information is retained in the IR for as long as it is required, rather than building it from scratch each time. Moreover, progressive lowering is strongly suggested. Rather than moving from the source to the target in a couple of transformations, the adoption of many different steps is suggested. This is a direct consequence of a philosophy followed by compiler designers for years: using a fixed amount of steps, and inserting each analysis/optimization within one of them.

- *Traceability.* Each operation should be traced back easily to the source code, for the compilation process to always be transparent. This is something required in the context of safety-critical or cryptographic applications, to obtain certifications stating that no security issue has been introduced by the compilation process. This problem is usually referred to as *What You See Is Not What You eXecute* [32].

2.3.2 MLIR Jargon

MLIR provides a textual representation for each extension; by keeping things coherent with standard formatting, parsing gets simplified.

The basic element of an IR is an *Operation*; this can be either an instruction, a function, or a module. Each user can define its operations in a declarative way, through a TableGen approach. Each operation is identified through a string stating both the name of the operation (`add`, `load`...) and its dialect (a collection of similar operations, such as `affine`, `std`...). An operation requires $N \geq 0$ *operands* and provides $M \geq 0$ results; each of these *values* is maintained in an SSA format. Every value also has a *type*, to be defined at compile time.

An example of an MLIR snippet, as reported in [17], is shown in Listing 2.3, implementing a polynomial multiplication with the `affine` and `std` dialect.

```
// Attribute aliases can be forward-declared.
#map1 = (d0, d1) -> (d0 + d1)
#map3 = ()[s0] -> (s0)
// Ops may have regions attached.
"affine.for"(%arg0) ({
    // Regions consist of a CFG of blocks with arguments.
    ^bb0(%arg4: index):
     //Blocks are lists of operations.
     "affine.for"(%arg0) ({
         ^bb0(%arg5: index):
         // Ops use and define typed values, which obey SSA.
         %0 = "affine.load"(%arg1, %arg4) {map = (d0) -> (d0)}
         : (memref<?xf32>, index) -> f32
         %1 = "affine.load"(%arg2, %arg5) {map = (d0) -> (d0)}
         : (memref<?xf32>, index) -> f32
         %2 = "std.mulf"(%0, %1) : (f32, f32) -> f32
         %3 = "affine.load"(%arg3, %arg4, %arg5) {map = #map1}
         : (memref<?xf32>, index, index) -> f32
         %4 = "std.addf"(%3, %2) : (f32, f32) -> f32
         "affine.store"(%4, %arg3, %arg4, %arg5) {map = #map1}
         : (f32, memref<?xf32>, index, index) -> ()
         // Blocks end with a terminator Op.
         "affine.terminator"() : () -> ()
         // Ops have a list of attributes.
     }) {lower_bound = () -> (0), step = 1 : index, upper_bound = #
        map3} : (index) -> ()
     "affine.terminator"() : () -> ()
 }) {lower_bound = () -> (0), step = 1 : index, upper_bound = #map3}
  : (index) -> ()
```

Listing 2.3: Example of MLIR representation.

Each operation also has some *attributes*, *regions*, *successor blocks*, and *location information*; some verifiers are attached as well, to guarantee the IR validation.

An *attribute* is compile-time information about operations, other than the name itself. Attributes are typed, and encoded in a dictionary. In Listing 2.3, attributes are used to express the lower bound, step and upper bound of the for loop. As it happens for operations, attributes can be defined by the users.

The *location information* is how the *traceability* principle is implemented, by moving information throughout conversions.

An instance of an operation can have a list of regions attached: a *region* provides the nesting mechanism of MLIR, as it contains many blocks, each of them containing operations (and so on, recursively). In each region, the blocks form a control flow graph, while the semantics of a region are defined by the contained operation - for instance, in `affine.for`, the block `bb0` is executed N times according to the boundaries.

Each block terminates with a *terminator*, which is a way to transfer the control flow to another block. ϕ-SSA values are encoded through block arguments rather than explicit operations so that a terminator is also in charge of providing these values for the successors.

The type system can be extended as well, while still requiring strict type equality checking; MLIR is structured into functions and modules. To simplify their implementation, they both are operations in the `builtin` dialect.

A *dialect* is a logical grouping of operations, attributes, and types under a single namespace. Through dialects, one can organize the ecosystem of a language and domain-specific semantics. In this way, it is possible to design modular libraries, integrating more than one dialect at a time depending on the required semantics.

Rather than putting too many different concepts into a single dialect, having many of them allows to limit the scope of the verifiers and new functionalities.

2.3.3 Main Dialects

While the users are free to define their dialects for their purposes, some of them are already provided by MLIR, especially to handle the first steps of the lowering process. What follows is a list of those involved in Dynamatic, as depicted in MLIR documentation [33].

- `affine` is the target of the frontend process; some control flow structures are not lowered completely, so that memory analysis can still be handled properly. Also, at this stage, polyhedral and loop transformations are still available.

- `scf` (*structured control flow*) contains operations that represent control flow constructs such as `if` or `for`. This is the logical successor of the `affine` dialect, and it is usually lowered down to the `cf` dialect. Being *structured*, no `goto` is present.

- `cf` (*control flow*) contains non-region-based control flow constructs, so that flow moves from one BB to another only through conditional (`cf.br`) and unconditional (`cf.cond_br`) branches.
- `arith` holds basic integer and floating point operations; this can also work with vectors and tensors.
- `func` contains the operations surrounding high-order function abstractions, such as call and return statements.

2.4 Dynamatic

Dynamatic is a research project initiated at the Processor Architecture Lab (LAP), at EPFL, in 2018, with a series of publications related to the idea of doing Dynamic High-Level Synthesis. This is opposite to the standard approach of doing static HLS, such as in Vitis™ by AMD. Before diving into the structure of the compiler, let's focus on the difference between the two approaches.

2.4.1 Static HLS vs. Dynamic HLS

An HLS tool is called *static* when the scheduling of the operations in the system is performed *statically* at compile time, without using the runtime information to improve the timing of the system.

This approach has the main advantage of ensuring predictability in the circuit while keeping the compile process as simple as possible. As reported in [5], this approach looks close to the VLIW way of composing long instructions at compile time, skipping anything that could resort to a hazard or lead to run-time problems.

It is not impossible to take into account the run-time information, implementing pipelining and parallelization of operations. This does not come, however, automatically: it requires some huge *tweaks* to the original controller of the circuit, possibly leading to a significant increase in area and power consumption.

To have an idea of how a static HLS works, let's consider the example in Listing 2.4, which is the same as reported in the official Dynamatic webpage [15].

```c
float d = 0.0;
int i;
for(i = 0; i < 100; i++){
    d = A[i] - B[i];
    if(d >= 0)
        s += d;
}
return s;
```

Listing 2.4: Dynamic scheduling example.

Let's suppose the following inputs are provided: `A[0] = 1, A[1] = 4, A[2] = 2, A[3] = 4` and `B[0] = 3, B[1] = 3, B[2] = 2, B[3] = 5`. This means that the addition at line 6 should be done only for the second and third iterations.

In a static scheduling of the operations, the window to perform the sum should always be allocated. If no sum is done, the corresponding time slot is just wasted rather than replaced with the following operations.

On the contrary, the strength of dynamic scheduling is to let the components themselves decide when to run (in jargon, *fire*) depending on the provided inputs. In the above case, as soon as the comparator from line `5` fires `false`, no operands will be provided to the adder from line `6`, thus the operation will not run. On the contrary, a new iteration can immediately start.

Together with this, since the computation of `s` is independent of the memory accesses and the subtraction in line `4`, it is possible to access the memories in `A` and `B` before current `s` is computed.

Figure 2.6, which is inspired by [15], shows the behavior with respect to timing. The static approach requires 17 cycles to provide a result, while the dynamic one only 12; this is almost a 30% improvement.

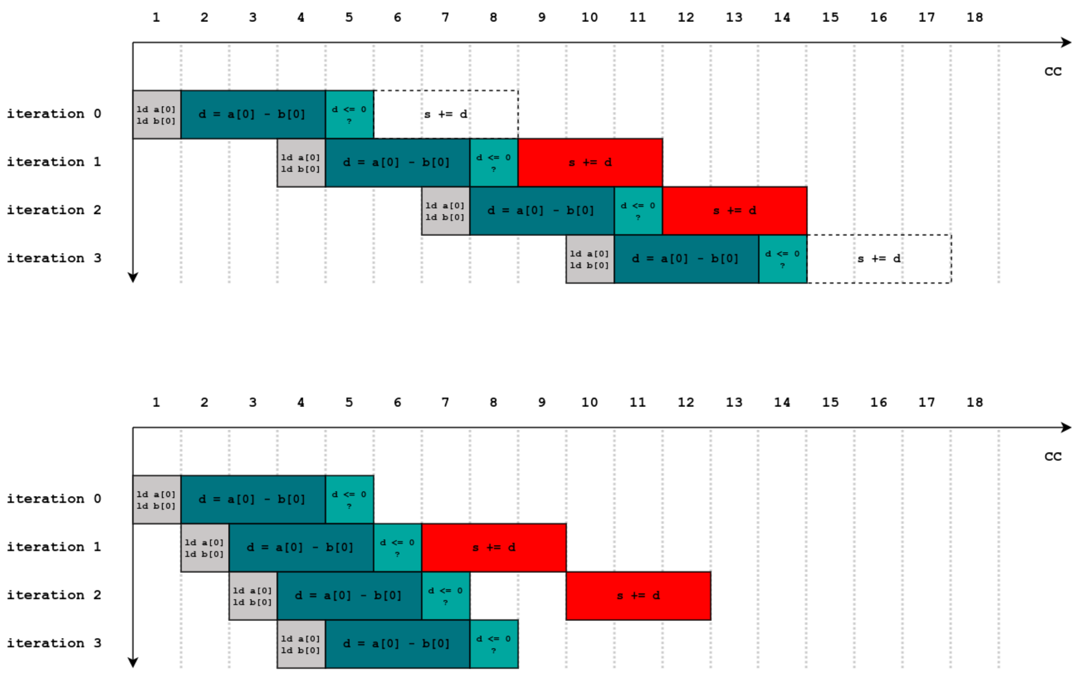


Figure 2.6: The diagram shows the differences between static (top) and dynamic (bottom) scheduling for Listing 2.4, inspired by the figure in [15].

2.4.2 Dataflow Circuits

The basic mechanism behind the construction of a dynamic scheduled circuit was first explained in [5] and [34]. The following section is inspired by a tutorial given by the authors in 2020 [35].

As explained above, the point of such circuits is to let the components communicate between themselves using a handshake protocol, without a controller which handles them. Such components are generally called *elastic*. This approach can be defined as *latency-insensitive*, and it is the usual foundation of *dataflow circuits*. The exchanged data is referred to as *token*.

The handshake is called *valid/ready*: the source component indicates token availability with 1-bit *valid*, the target component indicates the readiness to receive with 1-bit *ready*. Having this handshake logic with the associated circuits is the reason why, in general, dynamic scheduling requires more area than static scheduling. Usually, a component with N inputs (such as an adder) can run only when all the inputs are available at once. The inputs cannot be ready until the last computation is finished.

A token is said to *stall* when the source is *valid* but the destination is not *ready*; when both the signals are 1, then the exchange can happen; if the target is *ready* but no token is available, then it waits until the computation can start. It is quite common for a token to stall. For instance, imagine the operands of a non-pipelined divisor: no execution can start before the previous one is terminated, thus the inputs of the divisor are not *ready*.

The existence of this protocol between components justifies the lack of a controller, with the outcome of having a sort of *pure dataflow* circuit. It is convenient to also have some *control-only* signals, lacking data (they are only made of the two handshake bits). In this way, one can represent an abstract token for control reasons (for example, to notify a component of the fact that something happened in the circuit).

To guarantee correctness, two rules must be followed:

1. Each token produced must be used;

2. Each token produced must be used once and only once.

By default, the result of an adder cannot be used in multiple locations, since it would violate the second rule. The algorithms inside the compiler make sure that such restriction is always valid.

Each standard hardware component (registers, arithmetic units, memory controllers, logic gates...) has now a dataflow counterpart, which behaves identically to the original component while exchanging tokens (both on the receiving side and the sending side) with a *valid/ready* protocol. In addition to these standard components, specialized elements manage the correctness of the circuit, as shown in Figure 2.7:

- **Eager Fork** (`FORK`): Duplicates incoming tokens to multiple outputs. Tokens are sent as soon as any successor is ready; however, a new token can only be accepted after all successors consume the previous one. This component allows the re-usage of a token in multiple locations, since the producer has only one consumer (the `FORK` itself) while each of the N consumers is connected to one of the N `FORK` outputs.

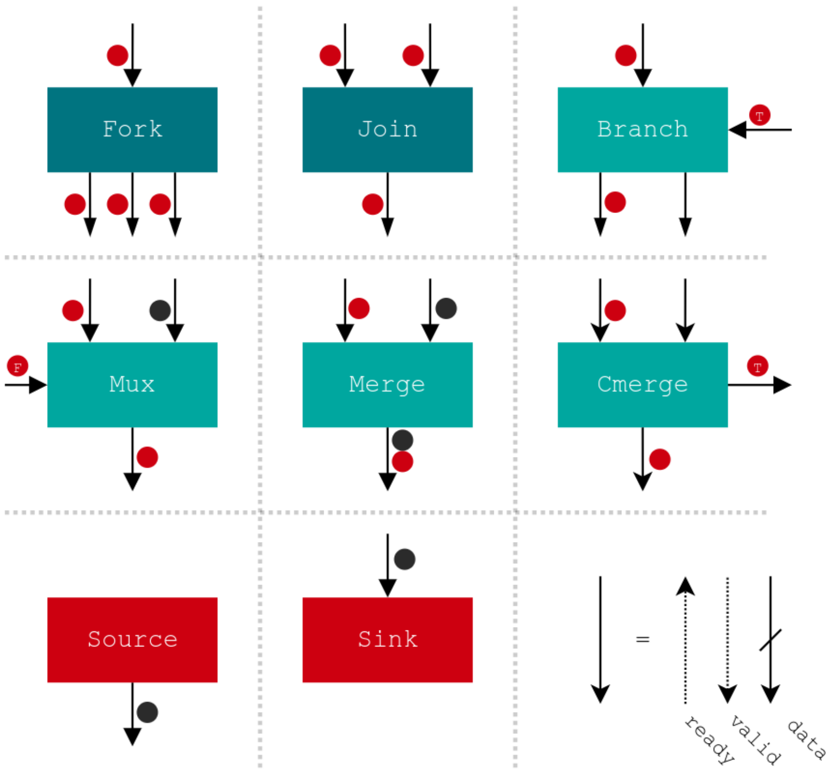


Figure 2.7: This list shows the main dataflow components, inspired from [35].

- **Lazy Fork** (`LFORK`): Similar to the eager fork but distributes tokens only when all successors are ready at the same time. This allows a form of synchronization between the successors so that they can run only at the same time.
- **Join** (`JOIN`): The output is provided only when all the inputs are valid. This is usually used with control-only tokens so that a valid signal is provided downstream only when all the inputs are valid too. It works as a synchronization mechanism, and it is in charge of synchronizing tokens in arithmetic units (an adder can provide a result only when its two inputs are available).
- **Merge** (`MERGE`): A component that forwards a token received from any input to its output. This can be seen as *non-deterministic* since it cannot be stated which input it is going to accept at any time, and it only depends on their availability at run-time.
- **Multiplexer** (`MUX`): A deterministic variant of merge, which forwards tokens based on a control input (choosing either the left or the right side of the multiplexer).
- **Control Merge** (`CMERGE`): Combines merge functionality with an additional output indicating which input was selected. It is useful to communicate to another component whether the left or right source was used.

- **Branch** (`BRANCH`): Routes tokens to different outputs based on a condition, implementing program control flow. As the condition is a 1-bit value, the available outputs are *true* or *false*.
- **Source** (`SOURCE`): Always valid to generate tokens for its successor. It works as an endless source of tokens, for instance, to have constants available.
- **Sink** (`SINK`): Always ready to consume tokens from its predecessor. This component is in charge of suppressing a token if it is not necessary, making sure that, as stated in the previous conditions, every token is consumed.
- **Buffer** (`BUFFER`): It works as a delay generator, accepting a token and providing it to the output after N cycles. This can also work as a *queue* made of M locations.

Since what was shown so far resembles the area of asynchronous circuits, it should be noted that it is fundamental to guarantee that no combinational cycle is present in the circuit. Buffers in dataflow circuits function similarly to registers in synchronous designs, ensuring that all combinational cycles are interrupted. Unlike standard registers, buffers can be placed on any circuit channel without affecting functionality (as a consequence of a *timing-insensitive* structure). The more buffers, the higher the area and the execution time. However, there are two points to consider:

1. **Critical Path:** Buffers can reduce the critical path by breaking long combinational delays. While the throughput might decrease this way (as a consequence of a higher latency), one might obtain a lower clock period during synthesis, leading to a lower execution time.
2. **Throughput:** Larger buffers, such as FIFOs, mitigate backpressure by storing tokens and enabling higher throughput whenever multiple paths have different processing time. This is a problem that usually arises in dataflow circuits, as shown in Figure 2.8. The `FORK` requires that all the tokens are delivered to the targets before a new token can be accepted at the input uphill. On the left figure, the first component can receive a token and start using it, while the component on the right cannot. For this reason, the right output of the fork remains *valid*, and its input is *not ready*. The component above the fork thus will not be able to fire at all, propagating the backpressure (*not ready* signal) uphill again. On the contrary, if a buffer is present, a token can be stored inside waiting to be used by the second component. In this way, the components above the fork are not backpressured.

Buffers should be put across the appropriate channels to increase performance while reducing the required number of buffers. Dynamatic optimizes buffer placement and sizing using a performance model that maximizes throughput at a given clock frequency [9]. Buffers are characterized by two properties:

1. **Transparency:** Indicates whether the buffer introduces sequential delay. Non-transparent buffers add a one-cycle latency, while transparent buffers act as pass-through elements. To simplify: in a transparent buffer, if no back-pressure comes

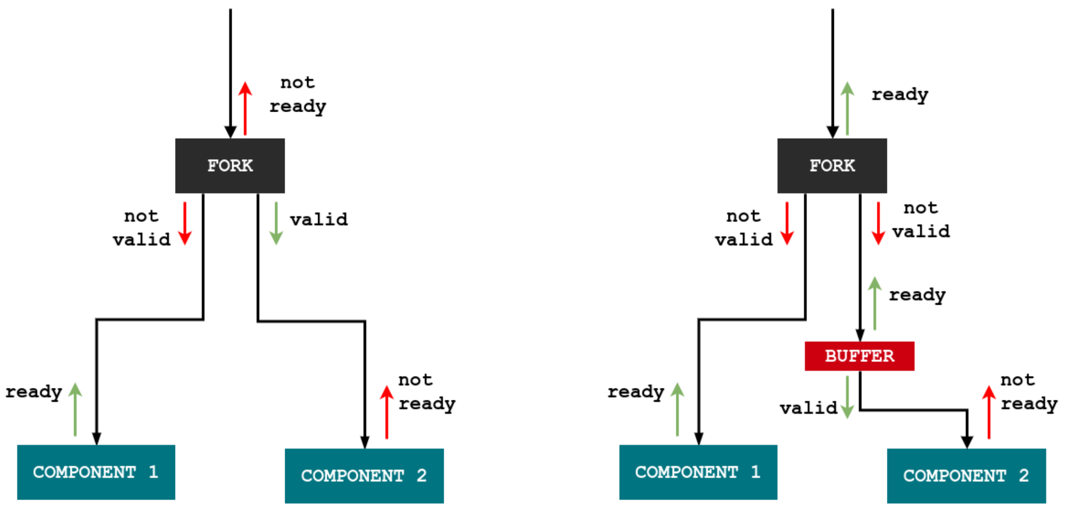


Figure 2.8: Backpressure happens each time the component downstream being *not ready* forces the previous components to stall. On the left, a situation without buffers is shown; on the right, the way buffers can solve the problem.

from the target, then the buffer is non-existent, otherwise, it stores the token. In general, non-transparent buffers (also called *opaque*) are used as little as possible to break sequential delays, while throughput (as shown in Figure 2.8) is optimized using transparent buffers.

2. **Capacity:** Refers to the number of slots available in the buffer for regulating throughput (the more tokens a buffer can accommodate, the less backpressure will be in the circuit and the higher the throughput).

Dataflow circuits interact with memories through load and store operations. This introduces the same kind of memory dependencies required by a CPU. In particular, if a load is after a write in the original sequence of the program, such a relationship needs to be respected at run-time as well.

When memory dependencies cannot be resolved at compile time, load-store queues (LSQs) [34] dynamically manage memory access, similar to their role in out-of-order CPUs. LSQs require explicit information about the original program order of memory accesses, provided through control-only paths.

This is also referred to as *cmerge network*: the components operate in the dataflow circuit in relation to their original CFG structure, and there is a control merge per BB firing when that same BB runs.

These paths ensure that tokens follow the correct execution sequence of BBs, allowing LSQs to resolve dependencies even with out-of-order accesses (more details will be provided in Subsection 2.4.4). Dynamatic simplifies LSQ design through compiler analysis, grouping non-conflicting memory accesses into separate queues (to reduce area overhead) and connecting conflict-free accesses to simpler interfaces. The main contributions to the area of the LSQ in dynamic HLS are [34] and [7]; with these techniques, area and critical path are enormously optimized with respect to a naïve approach having all the memory accesses handled by an LSQ.

2.4.3 A Dataflow Circuit Example

The best way to understand the functionalities and correctness of a dataflow circuit is by going through an example. In particular, the example in Listing 2.4 becomes the circuit in Figure 2.9.

This picture omits some details; however, it is complete enough to address the main characteristics of the method. The top-right `MERGE` is in charge of picking either the initial value of `i` or the updated value. The top-left `MERGE` is, on the contrary, in charge of handling the updated value of `s`.

Each time there is a `MERGE`, there is always a combinational cycle to handle. In this case, one cycle is made by `MERGE`, `BUFFER`, `ADDER`, `FORK`, and `BRANCH`. `BUFFER` (of type opaque, made of one slot) is then in charge of breaking such a loop.

Since `i` will be used in many locations, a `FORK` is required; on the left, it works as an index to access the memory locations `A` and `B`, while on the right it is incremented for the next iterations.

`d` is computed by subtracting the two loaded values; `s` is always computed as the sum of `d` and the previous `s`. The new value of `s` is selected depending on the comparator output.

Once `i` is computed, the same value is used to determine whether the iterations are done or not. In case the loop is finished, `BRANCH` is in charge of notifying this.

2.4.4 Dynamatic Flow

As described in Figure 2.1, a compiler is made of subsequent stages aimed at targeting different transformations; the same pipeline holds for Dynamatic. It is implemented in MLIR (Section 2.3), using both standard dialects and custom dialects (inspired by the CIRCT project [36]). A *transformation pass* modifies the IR while maintaining the same dialect; a *conversion pass* modifies the dialect too. This section lists all the passes involved in the main version of Dyanamtic as provided in [16] in December 2024.

- **Frontend**. Dynamatic utilizes *Polygeist* [37] as its frontend for processing C code. In the HLS context, Polygeist is in charge of performing memory analysis, disambiguating memory accesses that are considered *safe* (no Read-After-Write dependencies). This capability is crucial for optimizing dataflow circuits since, as stated before, having fewer conflicts implies fewer (or none) LSQs, thus decreasing their area overhead. The result of such a transformation is an IR in the *affine* dialect, as explained in Subsection 2.3.3.

- **Conversion and Transformation of `scf` dialect**. The `-lower-affine-to-scf` pass converts affine loops and operations into the `scf` dialect, which represents loops and conditionals in a more general form, closer to the CFG representation.

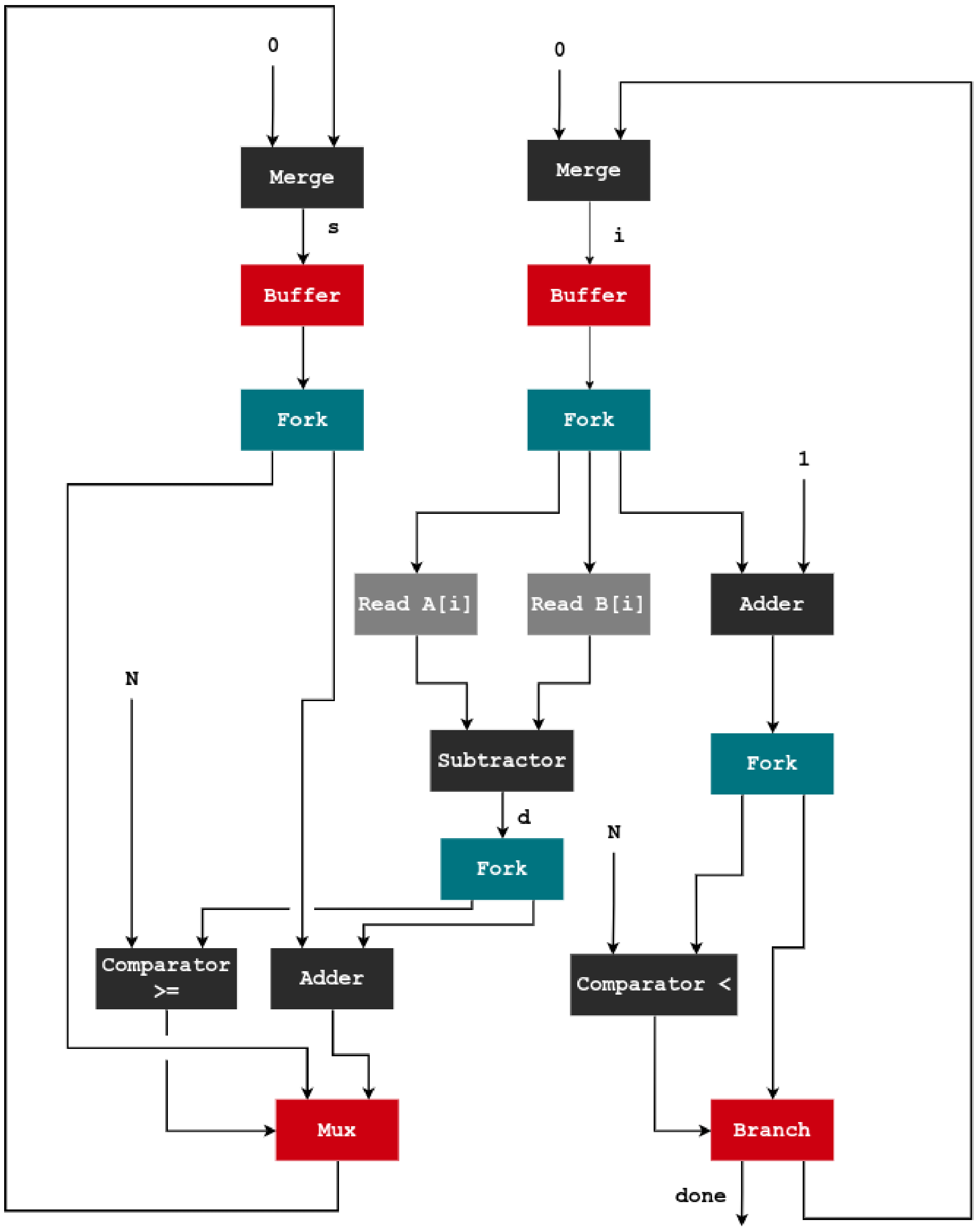


Figure 2.9: Example of dynamic scheduled circuit obtained from Listing 2.4.

The `-flatten-memref-row-major` pass reorganizes memory references into a row-major layout, simplifying memory access patterns, and ensuring compatibility with standard hardware memory models. The `-scf-simple-if-to-select` pass replaces simple conditional branches in `scf` with a select operation, which simplifies the control flow by avoiding explicit branching, improving efficiency and readability (this is the reason why Figure 2.9 shows a `MUX`). Lastly, the `-scf-rotate-for-loops` is in charge of transforming loops that are provably executed at least once into *do-while* loops. This last optimization reduces the number of BBs in the CFG, thus improving the overall area.

- **`cf` conversion**. When moving down to the `cf` dialect, some more control flow simplifications are performed, mainly getting rid of useless control flows. While many of these transformations are standard MLIR's, some are introduced for Dynamatic's objectives: `-arith-reduce-strength` is in charge of simplifying arithmetic operations whenever it is possible, for instance by substituting a multiplication in a sequence of shifts, additions and subtractions ($x \cdot 5 = x \cdot 4 + x = x << 2 + x$). `-push-constants` moves `arith::ConstantOp` operations to the block(s) using them, so that fewer values have to be moved through SSA nodes.

- **`handshake` conversion**. The `handshake` dialect was designed in the context of CIRCT [36] to represent a set of components interconnected via a *valid/ready* handshake. At this level of abstraction, the control flow mechanisms are removed, and a proper algorithm is used to obtain an IR that already resembles the final circuit in terms of interconnections between hardware components. Dynamatic exploits the algorithms presented in [5] and [35].

  As a starting point, the `cf` dialect represents operations within BBs, explicating both the control flow relationships and the SSA nodes for each BB.

  1. **SSA Maximization**. The first step of the conversion is to modify the `cf` structure so that each variable being a *live-out* for a BB can only be a *live-in* for a successor BB. This is in line with the way the circuit is supposed to be made, having a connection between two components only if these components are in adjacent BBs. This can be expressed with two conditions: each BB must send data only to the immediate successors; each BB must receive data only from its immediate predecessor. Since each non-necessary ϕ gate introduces an overhead in hardware resource and timing, this implementation choice can be considered as suboptimal; *Fast Token Delivery* [10], with its simplified connections depicted in Chapter 3, will get rid of this step.

  2. **Handling Control Flow**. Each live-in of a BB corresponds to a ϕ node in the SSA representation, and it is implemented as a `MUX`. If the BB has multiple successors, then a branch is required to move a token to the correct destination (i.e., the live-in of the subsequent block), otherwise, the connection is direct since it is unconditional. This approach ensures that the tokens are correctly moved between BBs according to the original CFG. Notice that, so far, the `MUX` does not have a control input yet. The next step of the flow will find such a value, and improved techniques will be provided in Chapter 3.

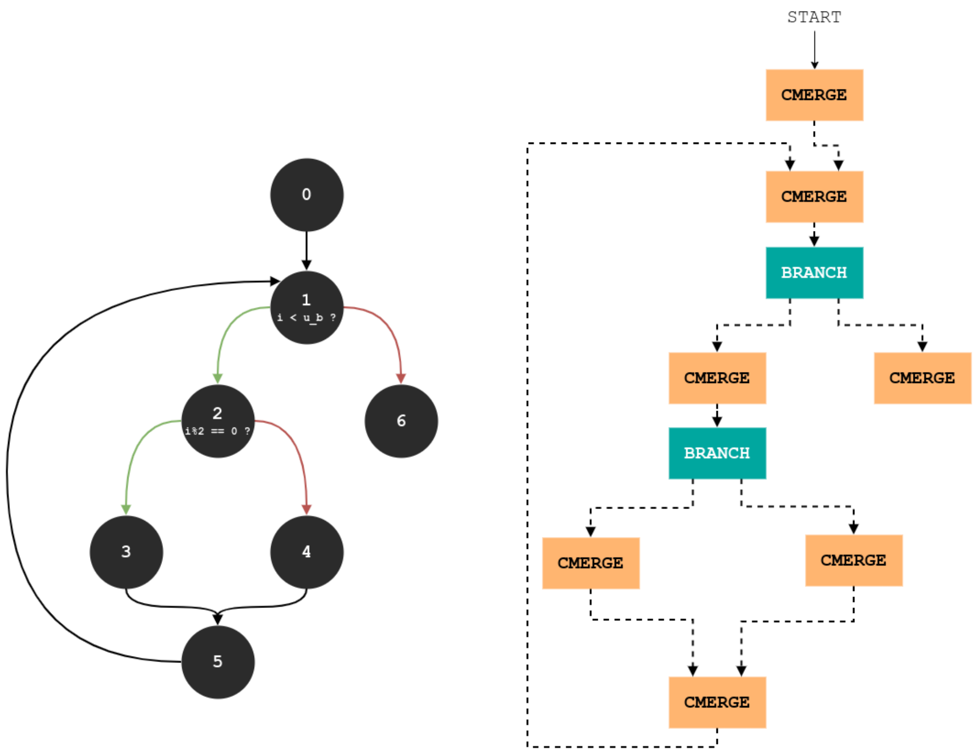


Figure 2.10: On the left, the same CFG from Figure 2.3 is reported; on the right, its corresponding *cmerge network*, mimicking the way the BBs execute. Each BB has a `CMERGE` with as many inputs as predecessor; if the branch is conditional, a `BRANCH` element is inserted to switch the flow correctly.

3. **In-Order Control Network**. Some operations, such as constant generation, do not require inputs but must still be triggered correctly: a constant, such as `N` or `0` in Figure 2.9, is produced only whenever it needs to be used. The `MUX`es obtained from the ϕ conversion also require a condition to know which BB was the predecessor. To address this, an in-order control path is introduced, representing a data-less signal that serves as a live-in and live-out for every BB. This is the *cmerge network*, in charge of signaling the execution of a BB. This network resembles the control flow graph, as Figure 2.10 shows. It is in charge of feeding `MUX`es and triggering constants; also, it is used in the interaction with the memories, as Chapter 4 will show. The *cmerge network* makes a dataless token circulate, mimicking a program counter.
4. **Building the Datapath**. After establishing the control flow, constructing the datapath becomes straightforward. Each instruction in the BB is mapped to a corresponding dataflow component. At this stage, the token counting is not handled yet, so no forks or sinks are instantiated. However, the memory controllers and the LSQs are created to allow interactions with the memory.

- **`handshake` Transformation**. The following transformations are applied in the flow to optimize the generated dataflow circuit:

  1. `-handshake-replace-memory-interfaces`. During the conversion process, LSQs are instantiated to handle potential memory conflicts. However, as shown in [7], not all conflicts need to be explicitly managed in a dataflow circuit due to the inherent movement of tokens within the circuit. This pass reduces the number of LSQs by removing unnecessary ones.
  2. `-handshake-minimize-cst-width`. By default, constants in the circuit are instantiated as 32-bit operands, even if their actual value requires far fewer bits. For instance, the constant value 5 only requires 3 bits to be represented. This pass is in charge of performing the bitwidth transformation.
  3. `-handshake-optimize-bitwidths`. Similar bit-width optimizations can be applied to arithmetic components. Initially, all arithmetic units are defined as 32-bit wide. However, if an arithmetic operation is only used in a context where smaller bit-widths suffice, this pass reduces the size of the components. For example, consider a comparator that compares the output of an adder with the constant value 5. If the constant only requires 3 bits, and the adder is used solely in this context, both the adder and comparator can be resized to 3 bits. This optimization significantly reduces area and improves timing without compromising functionality.
  4. `-handshake-materialize`. If a token is used in multiple places, it must be duplicated using `FORK` components. Conversely, if a token is unused, it must be discarded using `SINK` components. This pass is in charge of such transformation.

- **Buffering**. As explained in Subsection 2.4.2, the insertion of buffers is necessary both for correctness and to enhance throughput. Many techniques have been studied to achieve optimal results. While the first approach to buffering consisted of inserting buffers just to get rid of combinational cycles, [9] introduces a more clever way of solving the issue by adopting an ILP formulation.

- **`hw` Conversion**. The `hw` dialect is again a custom dialect that is in charge of being the last step towards the final circuit creation. Its components and connections are exactly the elements that need to be created in VHDL/Verilog (Dynamatic currently supports both), including also the corresponding entities and parameters. From here, the creation of hardware is almost a one-to-one translation.

In the official GitHub repository of the project [16], all the instructions to compile and run Dynamatic can be found.

Chapter 3

The *Fast Token Delivery* Methodology

3.1 Motivations

The example proposed in Chapter 2 is designed to show the benefits of the dynamic HLS approach over a static one. It is also valuable to consider which are the limitations of the methodology exposed in Section 3.2. The example in Listing 3.1 is taken from [10].

```c
int f(int* A, int* B, int size) {
    int x = A[0];
    int y = B[0];
    int i = 1;
    do {
        if (cond) {
            A[i] = x;
        } else {
            B[i] = y;
        }
        i += 1;
    } while (i < size);
    return x + y;
}
```

Listing 3.1: Kernel resulting in sub-optimal output in Dynamatic.

As it was stated before, although in a dataflow circuit the controller is distributed among components, there is still a *cmerge network* in charge of driving the sequence of BBs to execute according to the way they would work in the original flow.

The code in Listing 3.1 leads to the CFG and circuit in Figure 3.1. In the circuit, the control network is highlighted in orange; BBs are drawn in gray. For simplicity, some connections have been omitted (together with the content of `BB3`, which is specular to `BB2`).

The examination of Figure 3.1 shows that most of the components are only in charge of steering the flow, and not to perform computation. Such redundancy is a consequence

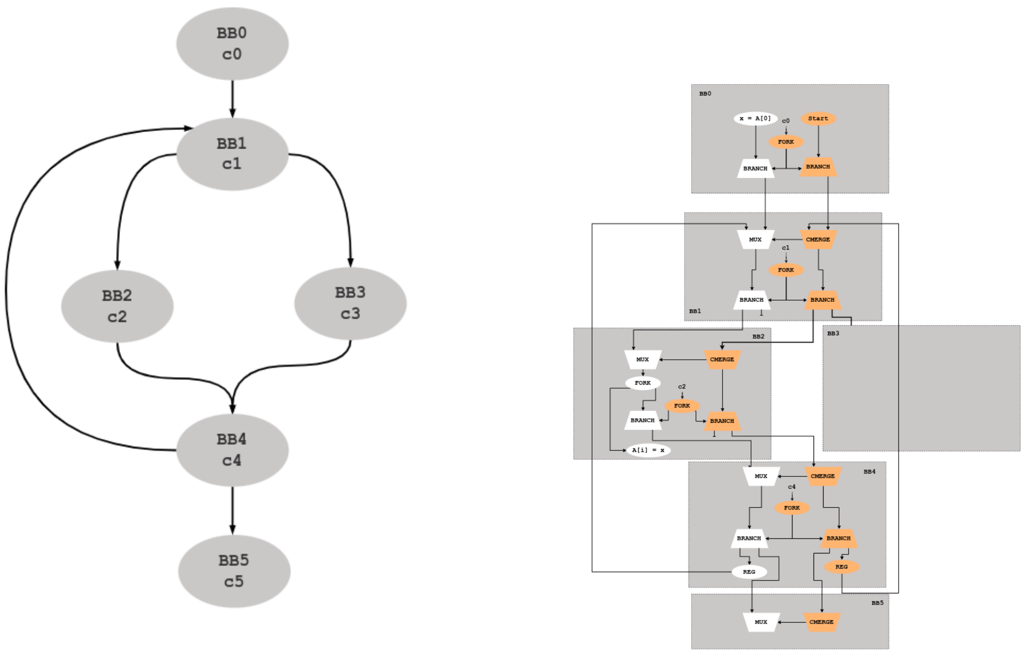


Figure 3.1: On the left, the CFG from Listing 3.1 is shown; on the right, the circuit which an be obtained using the standard Dynamatic approach.

of the way the circuit is built: all the BBs are maintained, all of them are triggered sequentially and tokens are forced to pass only to their direct successors, without any *shortcut*. In contrast, Figure 3.2 illustrates (with some omissions) a circuit that achieves functional equivalence. At a first glance, it is obvious that `x` and `y` are never updated after the first BB, thus their value can go directly to the sum in `BB5`. However, direct delivery of x from `BB0` to `BB2` would lead to runtime errors, since it depends on `cond` being true or false. As a consequence, x must traverse the branch to suppress the token if `cond` evaluates to `false`. While the introduction of a branch makes sure that no unnecessary token is delivered, it should be noted that only one token for `x` is normally created in `BB0`, while many of them are necessary due to the number of loop iterations. The leftmost part of the circuit in Figure 3.2 ensures that the x token is regenerated precisely as many times as `BB2` executes. The same holds for the right side, concerning y. The main point overall is to get rid of any redundant control structure and any explicit relationship with the CFG, to obtain a pure *dataflow* circuit (that is the reason why no BB is highlighted in Figure 3.2).

In this way, the area gets reduced, and tokens reach their destination faster, allowing for more parallelism. For instance, `BB5` can start the execution almost immediately after the circuit begins, well before the loop concludes, and thus all the components which use the result of the sum. The work in [10], called *Fast Token Delivery*, shows how to simplify the interconnection of components getting rid of the relationship with the BBs, as depicted above. This chapter will present the methodology, highlighting all the technical steps required to implement it in MLIR Dynamatic.

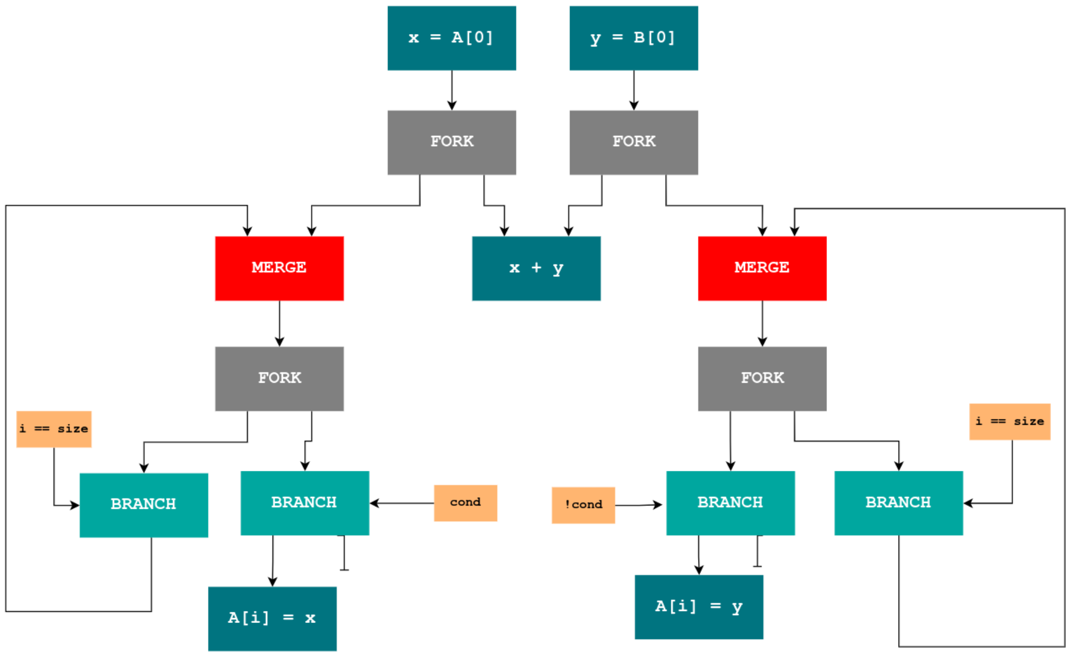


Figure 3.2: The circuit shown here results from Listing 3.1 when adopting an optimized token delivery strategy, as presented in [10].

3.2 The Algorithm

With respect to the explanation in Subsection 2.4.4, this algorithm represents a way to go from the `cf` MLIR dialect to the `handshake` dialect. This section explains from a broad point of view the general methodology which allows to obtain a correct circuit, while some more implementation details (related to the codebase itself) are provided in Section 3.4. The content of this section is largely inspired by the original explanation in [10].

3.2.1 `cf` to `handshake` Conversion

The depicted methodology starts without taking into account loops, thus having only forward edges in the control flow graph. Later, Subsection 3.2.2 will reintroduce them. This algorithm requires the GSA representation of the circuit as a prerequisite, so that every ϕ node has a corresponding μ or γ gate (Section 2.2). This representation is now given for granted; however, the full methodology to obtain such conversion is shown in Section 3.3. Having only forward edges implies having no μ gates (loop headers).

From Basic Blocks to Subcircuits

The first objective is to convert the operations of individual BBs into dataflow components and to interconnect components within the block itself. As it is already one in the original algorithm, [5] this stage is straightforward, since each instruction in the BB translates directly into a dataflow component, and any data-dependent parts are directly connected.

The blocks also include some γ nodes. These are translated to `MUX`es having two inputs and a one-bit condition. Multiple γs might be connected in a tree fashion, as it will be explained later.

Connecting Subcircuits

As each BB has now been covered - together with its internal data dependencies - the data dependencies across subcircuits must be handled. Notice that, according to Subsection 2.4.4, in standard Dynamatic, due to the *SSA Maximization*, two components can exchange tokens across BBs only if their corresponding BBs are adjacent in the CFG. The *SSA Maximization* is in charge of this process. In *Fast Token Delivery*, the final goal is to get rid of this redundant step, thus a more general system should be utilized.

The methodology is to consider each pair of producer and consumer, and generate some control logic to make sure that the token count is always matching. In particular, it will be necessary to suppress a token if the producer fires while the consumer does not (i.e., when the control flow suggests it will not execute). By considering each pair independently, this might lead to some redundant components. A final peephole optimization to run at the end of the conversion is in charge of simplifying the structure (Subsection 3.4.5).

It might also happen, on the contrary, that a consumer requires a token from a producer which did not fire. In this case, a dummy token is generated. This is almost nonsensical for data components (an adder cannot use a random value to perform an addition, otherwise the behavior of the circuit ends up being incorrect); however, it might be useful when considering delivering control tokens (Figure 3.5).

For each pair of producer and consumer, two new components are possibly added, as shown on the left of Figure 3.3.

After each producer, a `SUPPRESS` component is inserted to eliminate a token if its correspondent control input is true. This is implemented, as shown on the top-right of the picture, using a branch having the *true* output connected to a sink.

Before each consumer, a `GENERATE` component will inject a new token (arbitrarily either 0 or 1) each time its control input receives a true token, while the original token passes when the control input is false. This is a `MUX` which is connected to a source on the left side. The suppression and generation condition (indicated through f_{supp} and f_{gen}) must behave in a way that the functionality of the circuit is always correct. Also, if f_{supp} is computed as false, the `SUPPRESS` component can be omitted.

All producer-consumer pairs are treated generally, computing the control signal for `GENERATE`, which, in data-dependent scenarios, remains always false. When the algorithm perceives this situation, the `GENERATE` block can be directly omitted, or later optimized away. The following subsection shows how to determine f_{supp} and f_{gen}.

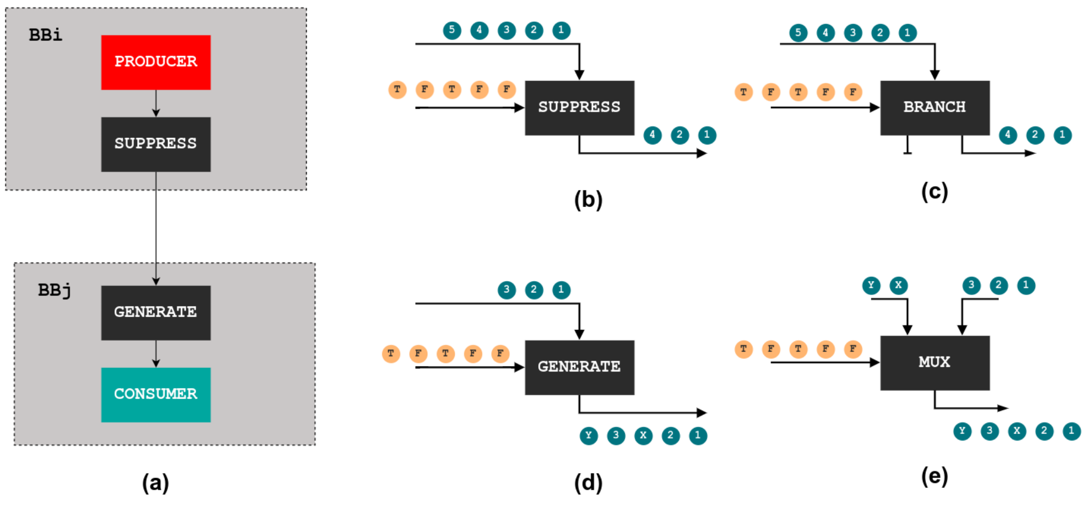


Figure 3.3: Suppress and Generator Components, as shown in [10].

Generating and Suppressing Tokens

The methodology will be explained both in a general way, and also by referring to Figure 3.4 as a real reference. This example is the same as used in [10]: the value of `x` must be sent from possible producers in `BB2` or `BB3` to a consumer in `BB4`, taking care of the conditions of each block.

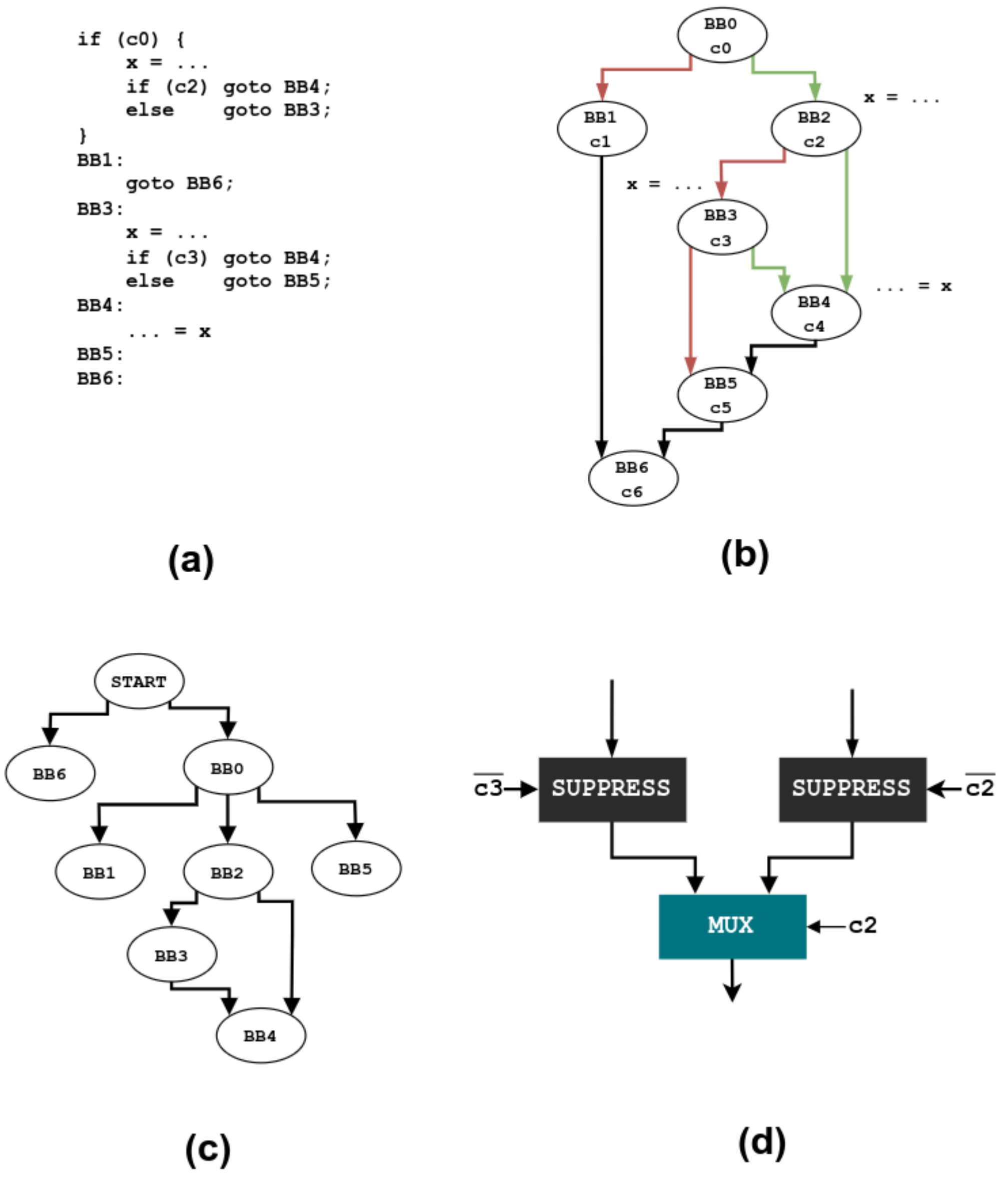
1. **Identify Control Dependencies.** Using the control dependence graph (Section 2.1), find S_{prod} and S_{cons}, the BBs on which, respectively, the producer and the consumer BBs are control-dependent. In the previous example:

   - (Producer in `BB2`, Consumer in `BB4`) yields $S_{\text{prod}} = \{\texttt{BB0}\}$ and $S_{\text{cons}} = \{\texttt{BB0}, \texttt{BB2}, \texttt{BB3}\}$.
   - (Producer in `BB3`, Consumer in `BB4`) yields $S_{\text{prod}} = \{\texttt{BB0}, \texttt{BB2}\}$ and $S_{\text{cons}} = \{\texttt{BB0}, \texttt{BB2}, \texttt{BB3}\}$.

2. **Eliminate Common Control Ancestors.** Remove $S_{\text{prod}} \cap S_{\text{cons}}$ from each set. The execution of these common ancestors impacts both the producer and the consumers, thus they should not be taken into account to know if only one of them ends up executing. Thus:

   - (Producer in BB2, Consumer in BB4) becomes $S_{prod} = \emptyset$ and $S_{\text{cons}} = \{\text{BB2}, \text{BB3}\}$.
   - (Producer in BB3, Consumer in BB4) becomes $S_{\text{prod}} = \emptyset$ and $S_{\text{cons}} = \{\text{BB3}\}$.

3. **Compute Conditions of Production and Consumption.** f_{prod} is defined as the Boolean expression describing when the producer releases a token; f_{cons} depicts when the consumer receives a token. To find these expressions, the CFG is traversed from each BB in S_{prod} toward the producer, and from each BB in S_{cons}



```
if (c0) {
    x = ...
    if (c2) goto BB4;
    else    goto BB3;
}
BB1:
    goto BB6;
BB3:
    x = ...
    if (c3) goto BB4;
    else    goto BB5;
BB4:
    ... = x
BB5:
BB6:
```

Figure 3.4: Delivery Problem as shown in [10]. The picture *a* shows he original code; *b* shows its CFG; *c* shows the control dependency graph; *d* the suppress mechanism obtained in the end of the algorithm.

to the consumer. Each path is a product of basic conditions, and those products are summed over all paths. If a control dependence set is empty, the resulting expression is true. In the example:

- (Producer in BB2, Consumer in BB4): $f_{prod} = 1$. $f_{cons} = c_2 + (\overline{c_2} \cdot c_3)$.
- (Producer in BB3, Consumer in BB4): $f_{prod} = 1$. $f_{cons} = c_3$.

4. **Adjust for MUX Select Signals.** If the consumer is a MUX, f_{cons} is multiplied by either f_{sel} or its complement, so that consumption only takes place if the input from this producer is chosen. Also, this additional step should be made only if $f_{cons} \cdot f_{sel}$ can be nonzero and if the producer's BB is not control-dependent on the BB whose condition drives f_{sel}. This should happen only if $f_{cons} \cdot \overline{f_{sel}}$ can be nonzero, and if the producer's BB is not control-dependent on the BB whose condition drives f_{sel}. In the example, the consumer is a `MUX`, since it needs to choose between the value from `BB2` and `BB3`:

   - (Producer in `BB2`, Consumer in `BB4`): $f_{sel} = \overline{c_2}$, $f_{prod} = 1$, $f_{cons} \cdot f_{sel} = (c_2 + \overline{c_2} \cdot c_3) \cdot \overline{c_2} = \overline{c_2} \cdot c_3$, which is nonzero. So f_{cons} is updated to $c_2 \cdot (c_2 + \overline{c_2} \cdot c_3) = c_2$.
   - (Producer in `BB3`, Consumer in `BB4`): $f_{sel} = c_3$, $f_{\text{prod}} = 1$ and $f_{\text{cons}} = c_3$. As `BB3` depends on `BB2`, f_{cons} should not be modified.

5. **Compute f_{supp} and f_{gen}.** Given the aforementioned definition, a token is suppressed if the producer executes but the consumer does not; a token is generated if the producer does not execute but the producer does. This leads to the following definitions: $f_{supp} = f_{prod} \cdot \overline{f_{cons}}$ and $f_{gen} = \overline{f_{prod}} \cdot f_{cons}$. Our example yields:

   - (Producer in `BB2`, Consumer in `BB4`): $f_{\text{supp}} = 1 \cdot \overline{c_2} = \overline{c_2}$, $f_{\text{gen}} = \overline{1} \cdot \overline{c_2} = 0$.
   - (Producer in `BB3`, Consumer in `BB4`): $f_{\text{supp}} = 1 \cdot \overline{c_3} = \overline{c_3}$, $f_{\text{gen}} = \overline{1} \cdot c_3 = 0$.

The boolean tokens that drive `SUPPRESS` and `GENERATE` must have values f_{supp} and f_{gen}, respectively. For relatively simple cases, as in the example, this already leads to a valid circuit shown in Figure 3.4.d. In general, there are some additional points needed to correctly produce these tokens, covered in the next subsection.

Delivering Control Tokens

Consider the CFG in Figure 3.5, representing a nested if-then-else program. Tokens for x and y generated in `BB0` and consumed in `BB2` result, by using the algorithm, in $f_{supp} = \overline{c_0} \cdot \overline{c_1}$. One might try a direct dataflow circuit to compute $\overline{c_0} \cdot \overline{c_1}$ using a simple gate (e.g., a dataflow NOR or NAND).

Although it appears logical to let f_{supp} be false if c_0 or c_1 is true, a hidden problem emerges. Consider the following truth table:

c_0	c_1	f_{supp}
false	false	true
false	true	false
true	token not produced	?

If c_0 is true, `BB1` does not run and so c_1 never appears, causing deadlock since the logic gate never sees both inputs and cannot generate the (expected) false token.

One naive workaround is to adopt fully dataflow logic gates to represent this Boolean function while treating c_0 and c_1 as standard producers and consumers of tokens. In this case, f_{gen} would result in a nonzero Boolean variable, thus having a generate block that

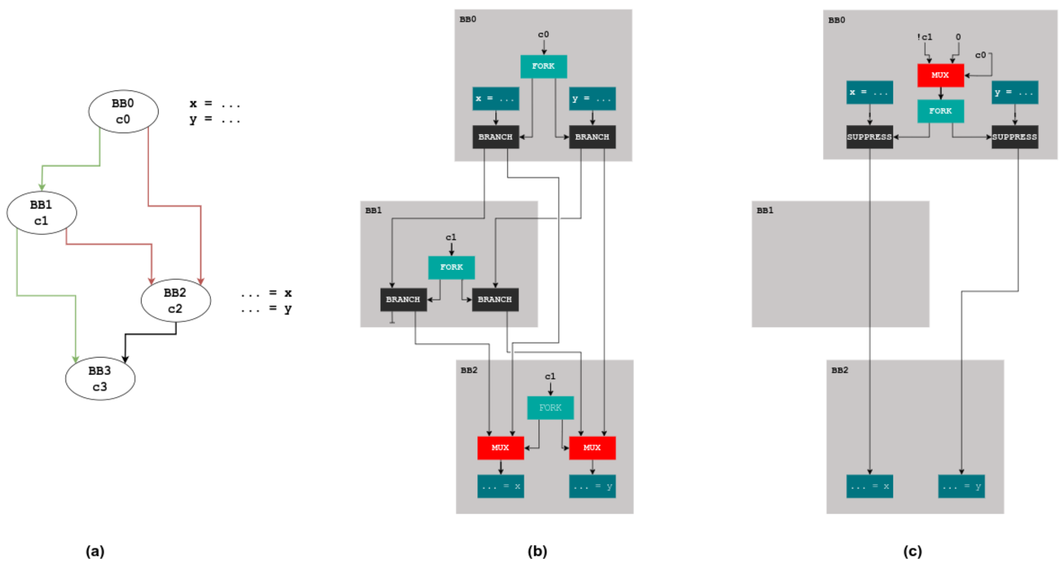


Figure 3.5: Example of problems in delivering control tokens, as shown in [10]. Picture *a* shows the original control flow graph; *b* depicts a functional circuit that is not minimized in area; the optimized version is shown in *c*.

inserts a dummy token (considering that, whichever the value of c_1 is, the resulting value will always be false). Although functionally sound, this method produces useless logic. A simple method is about implementing the logic operation $\overline{c_0} \cdot \overline{c_1}$ in a way that the input from `BB1` is ignored in c_0 is true.

Given $f_{supp} = f(l_0, ..., l_n)$, with l_i being a literal in the Boolean expression, here are the steps:

1. **Partial Ordering of Literals.** Build a graph G_{ord} whose nodes are the literals l_i of f, with edges from one literal to another if the corresponding BB in the Control Dependency Graph indicates a control path from the first literal's BB to the second's. For the example in Figure 3.5 there are two nodes, related to c_0 and c_1, and an edge $c_0 \to c_1$.

2. **Ordering of Literals.** Repeatedly take any literal l in G_{ord} that has no incoming edges, remove it, and add it to the ordered list L_{ord}. Once the first literal is chosen, the corresponding token is guaranteed to be present. For the example, the only possibility is $L_{\mathrm{ord}} = \{c_0, c_1\}$.

3. **Successive Shannon Expansions.** Assume the literals in $f(l_0, \ldots, l_n)$ are in the order of L_{ord} (if not, rename them). Write

$$f = \mathrm{MUX}(l_0,\ f_{l_0}(l_1, \ldots, l_n),\ f_{\overline{l_0}}(l_1, \ldots, l_n)) = \ldots$$

Then repeat for the remaining literals. This yields a chain of MUXes that implement f. The main benefit of this implementation is that unselected MUXes do not consume tokens, thus there is no deadlock. In the example, $f_{\text{supp}} = \overline{c_0} \cdot \overline{c_1}$ becomes $\text{MUX}(\overline{c_0},\ \overline{c_1},\ 0)$.

Without using the above methodology, the functional circuit is shown in the middle of Figure 3.5; on the contrary, the approach leads to the simplified version on the right of Figure 3.5.

3.2.2 Token Delivery With Loops

If the control flow graph has some loops, there are three new problems to handle:

1. Among the gating GSA functions, there are some μ gates in charge of handling loop-carried dependencies;
2. A token might be produced once outside of a loop, but employed multiple times within a loop;
3. A token might be produced multiple times in a loop, but employed only once outside of the loop.

An example is in Listing 3.1, where x is read in various loop iterations.

Controlling MUXes Within Loops

A μ gate becomes a MUX; however, the problem is understanding which condition drives such MUX. For the first iteration, the MUX should take x from outside the loop. In subsequent iterations, it should accept the local (updated) x until the loop completes. More generally, for nested loops, each time the outer loop repeats, the MUX should switch to the external value at the start and then continue with the local value for the inner loop. It seems then simple to adopt, as a control signal, the loop exit condition, so that the right input (regenerated) is picked only when it is known that a new iteration needs to run. The first iteration is a special case: the loop exit condition has not yet been computed, so the MUX must be forced to choose the external value.

To capture this behavior, an INIT component is adopted, that sends a token before the stream of loop conditions arrives (Figure 3.6). This insertion ensures that the loop selects *external* the first time and then, for each iteration, uses the loop condition to decide whether to keep taking the internal value or exit to the external one. As shown in Figure 3.6, a naive implementation with a MERGE can safely lead to the expected behavior.

Hence, for each μ-function, a simple graph analysis can determine the exit condition of its corresponding loop; then, a INIT component is instantiated, fed by such condition, and connected to a MUX.

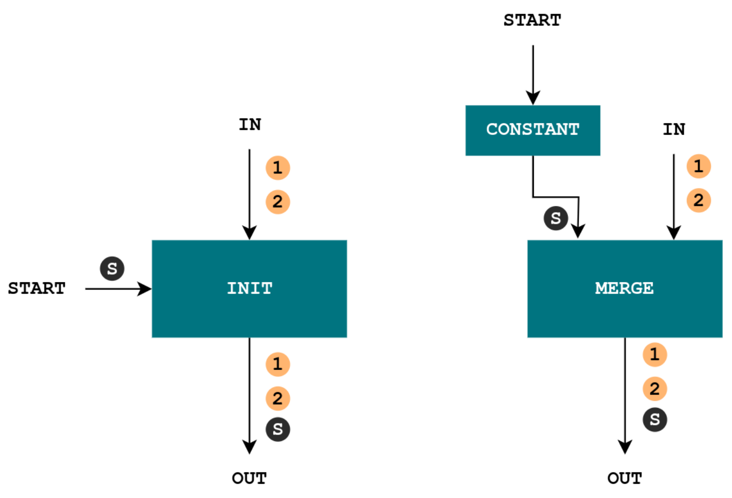


Figure 3.6: The `INIT` component and its behavior is shown on the left; on the right, a reasonable implementation using dataflow components only.

Producers Inside Loops and Consumers Outside

By default, if a producer is located in a loop and a consumer outside, there is a mismatched token count. Only the final token (from the last iteration) should be delivered to the consumer. This is a normal producer and consumer relationship, in which the suppression mechanism is driven by the loop exit condition: if the loop is finished, the token should not be suppressed. An example can be seen in Figure 3.7

Producers Outside Loops and Consumers Inside

The reverse issue arises when the producer is outside a loop and the consumer is inside it. In this scenario, a single token is generated, but it needs to be used repeatedly across multiple iterations. Once this pattern is detected with a simple loop analysis, the standard GSA loop-carried dependency transformation is employed, inserting $x = x$ right after its consumption. This leads to a second producer inside the loop, so there are two definitions for the same variable, requiring `MUX` logic in the loop header, as previously described. This arrangement then regenerates the token for as many iterations as needed. `SUPPRESS` nodes will prevent further regeneration once the loop finishes. An example of such an approach can be seen in Figure 3.8.

3.3 GSA Implementation

As previously mentioned, to run *Fast Token Delivery* it is necessary to have the GSA representation of the CFG. The `cf` dialect in MLIR already provides the ϕ gates in the SSA format; for this reason, a translation is required.

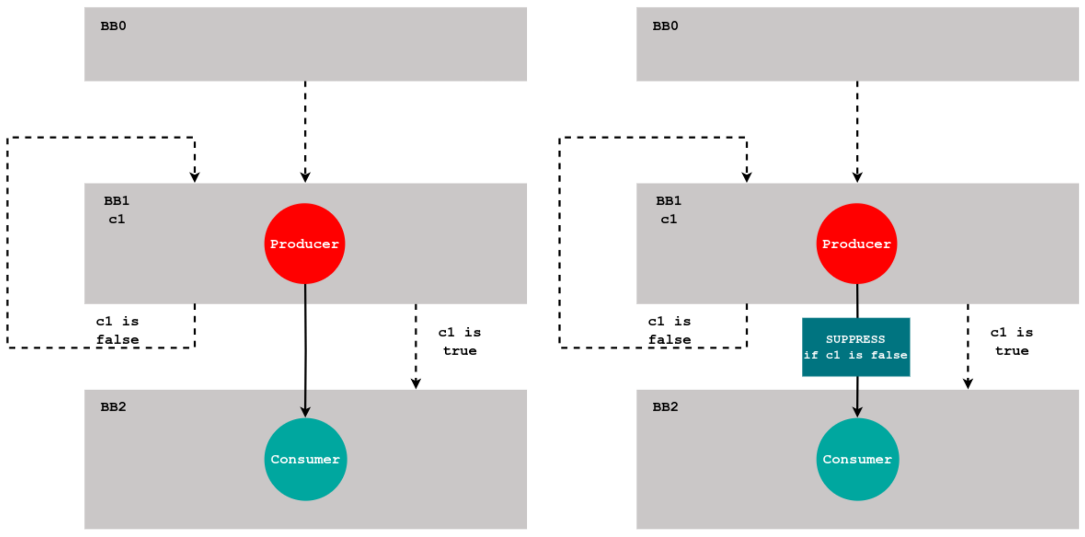


Figure 3.7: Example of producer inside a loop and consumer outside the loop: the IR on the left becomes the IR on the right once the SUPPRESS block is inserted.

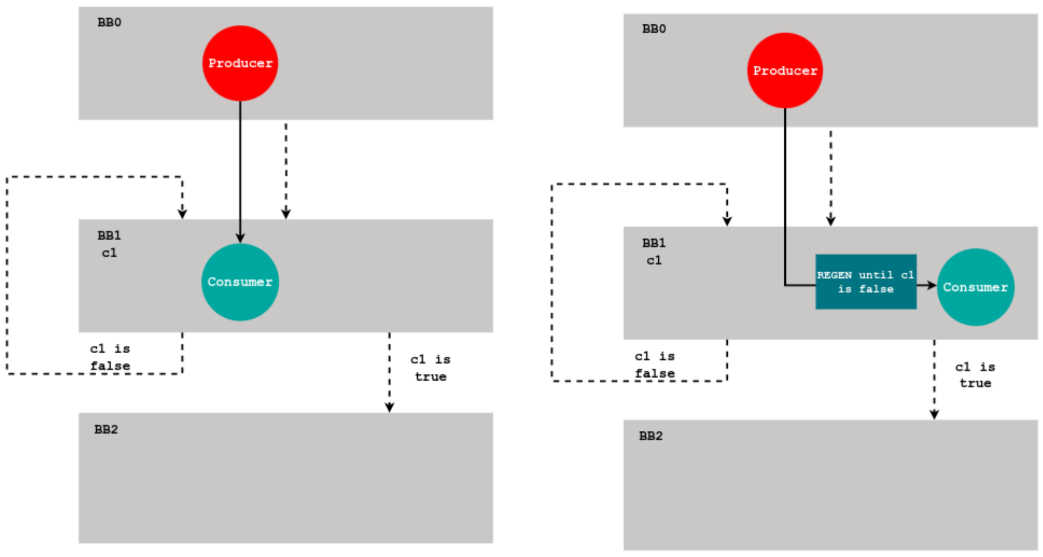


Figure 3.8: Regeneration mechanism for producers outside of a loop and consumers inside.

In this process, every block argument must be converted into either a γ function or a μ function. The γ function is a two-input MUX, each driven by a condition (a single ϕ might require multiple γ in a tree shape); the μ function has a start input to initiate the loop and an update input.

Various methods for constructing such functions have been discussed in the literature, including those presented in [29] and [23]. The method described by Havlak begins from a pre-SSA construction, limiting its utility since the SSA representation is already available in this context. The algorithm proposed by Tu [23], on the other hand, generalizes GSA construction to handle any complex control flow graph structure. The algorithm presented here is heuristic and tailored to achieve similar results. While formal correctness cannot be guaranteed, it has been confirmed to produce functionally correct results across all tested cases.

3.3.1 Algorithm for Constructing MU Functions from PHI Functions

The following conditions determine whether a ϕ function can be converted into a μ function:

- The ϕ function must have exactly two inputs.
- The ϕ function must reside within a loop.
- The ϕ function must be located in a BB that serves as the loop header.
- One input to the ϕ function must originate from outside the loop.

If these conditions are satisfied, the ϕ function is converted into a μ function.

In the example of Figure 3.9, all the above conditions hold, so it will be picked as a μ. This gate depends on the exit condition of the loop, c_1 in the example.

3.3.2 Algorithm for Constructing GAMMA Functions from PHI Inputs

All remaining ϕ functions must be converted into γ functions. However, a single γ is just a two-input `MUX`, which cannot express a multi-input ϕ. The strategy involves creating a *tree of* γ *functions*, each driven by a simple condition. The following steps outline the process for constructing such a tree for any ϕ function.

Initialization and Input Structure

Each ϕ_i, located in B_{ϕ_i}, is analyzed individually. The function ϕ_i receives $N \geq 2$ inputs, labeled as $i^0_{\phi_i}, i^1_{\phi_i}, \ldots, i^j_{\phi_i}, \ldots, i^{N-1}_{\phi_i}$. Each input originates from a BB $B^j_{\phi_i}$, indexed by j. All these blocks share a common dominator ancestor in the control flow graph, referred to as B_{CA}.

Input Ordering

The inputs of the ϕ_i function are sorted based on the relationships of their originating BBs. The indices are assigned such that if BB B_i properly dominates B_j, then $i < j$. This order does not affect the semantics of this original ϕ_i, since this gate is, by definition, order-less. However, it helps while moving on in the algorithm.

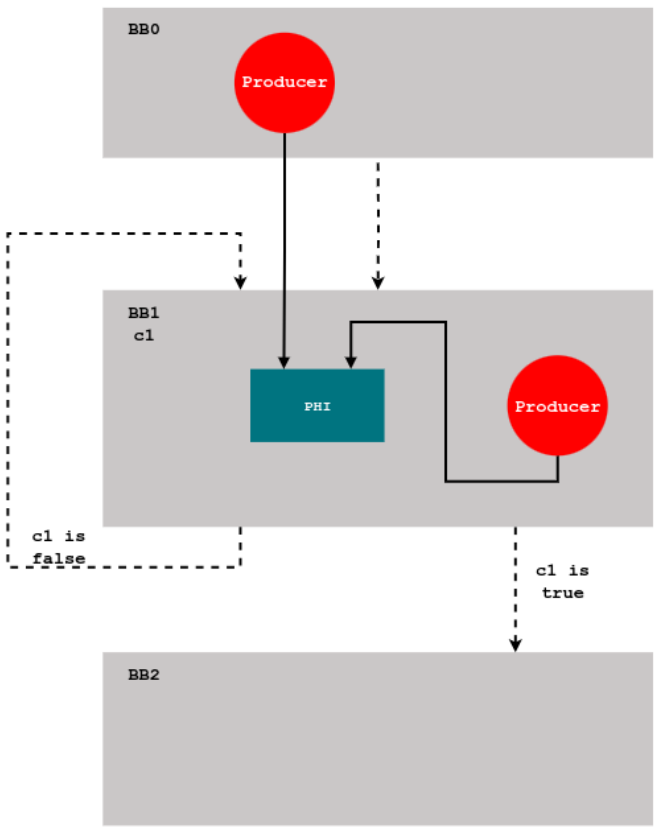


Figure 3.9: Example of ϕ which will be translated to a μ.

Path Identification and Boolean Conditions

For each input $i^j_{\phi_i}$, the path with the following characteristics are identified:

1. A path originates from B_{CA} and terminates in B_{ϕ_i}.
2. Each path must pass through $B^j_{\phi_i}$ but must avoid $B^k_{\phi_i}$ if there exists an input of ϕ_i in $B^k_{\phi_i}$ such that $k > j$.

Each path P_i is associated with a set of M boolean conditions, $c^0_{P_i}, c^1_{P_i}, \ldots, c^{M-1}_{P_i}$, representing what needs to happen in the CFG for that path to be taken. The boolean condition for a single path is expressed as:

$$c_{P_i} = c^0_{P_i} \wedge c^1_{P_i} \wedge \cdots \wedge c^{M-1}_{P_i}.$$

The boolean condition for the input $i^j_{\phi_i}$ is the disjunction of the conditions for all paths associated with it:

$$c^j_{\phi_i} = \bigvee_i c_{P_i}.$$

Constructing the GAMMA Function via Recursive Expansion

The γ functions for ϕ_i are constructed using a recursive expansion process, which considers one literal at a time:

1. Begin with the literal with the lowest index. Use this condition as the selector for the first γ function.
2. Divide the remaining boolean expressions into two sets:
   - Expressions where the literal is true; replace the literal with a true; this will become the *true* input of γ.
   - Expressions where the literal is false; replace the literal with a false; this will become the *false* input of γ.
3. Apply the expansion recursively to both sets, proceeding with conditions in increasing order.
4. Each γ function has two inputs: one is the γ obtained by recusively expanding the *true* input, while the other is the γ from the *false* input. If only one value remains in a branch, it is used directly as the argument of the γ function without further expansion.

Examples

Example 1: Simple Boolean Conditions Figure 3.10 shows a CFG with a value of `x` modified in two points, both in `BB1` and `BB3`. The algorithm from the previous example will be used to obtain a tree of γs in place of ϕ.

The inputs of the ϕ are located in `BB1` and `BB3`. Their common dominator ancestor is `BB1`. The ordering of the indices is already correct, since `BB1` dominates `BB3` and $1 < 3$.

There are two paths which allow to go from `BB1` to `BB4` without going through `BB3`: `BB1 - BB4` and `BB1 - BB2 - BB4`. The first path is covered with the conditions c_1, while the second is covered with $\overline{c_1} \cdot c_2$. For this reason, the input from `BB1` is used when the following boolean expression is true: $c_1 + \overline{c_1} \cdot c_2$. For the other input, the boolean condition is, on the contrary, $\overline{c_1} \cdot \overline{c_2}$.

Given the following input conditions:

$$V_1 \to c_1 + \overline{c_1} \cdot c_2,$$
$$V_3 \to \overline{c_1} \cdot \overline{c_2}.$$

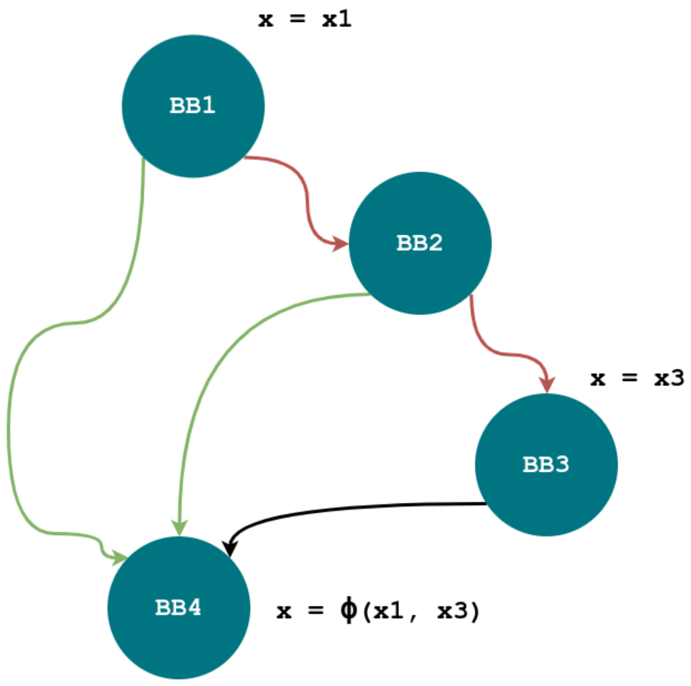


Figure 3.10: Example 1 of γ algorithm.

The γ tree is constructed as follows:

1. Start with c_1. When c_1 is true, only V_1 remains, so it is the only available value, connected to the true input of γ;
2. When c_1 is false, expand using c_2. If c_2 is true, V_1 remains; if c_2 is false, V_3 remains.

The resulting γ function is:

$$\gamma(c_1, V_1, \gamma(c_2, V_1, V_3)).$$

Example 2: Nested Conditions The same algorithm can be used to expand the ϕ in Figure 3.11. It should be simple to use the path between the common dominator of each node which defines `x` (`BB1`) until the ϕ BB (`BB8`) to obtain the following conditions:

$$\begin{aligned} V_4 &\to c_1 \cdot c_2, \\ V_5 &\to c_1 \cdot \overline{c_2}, \\ V_6 &\to \overline{c_1} \cdot c_3, \\ V_7 &\to \overline{c_1} \cdot \overline{c_3}. \end{aligned}$$

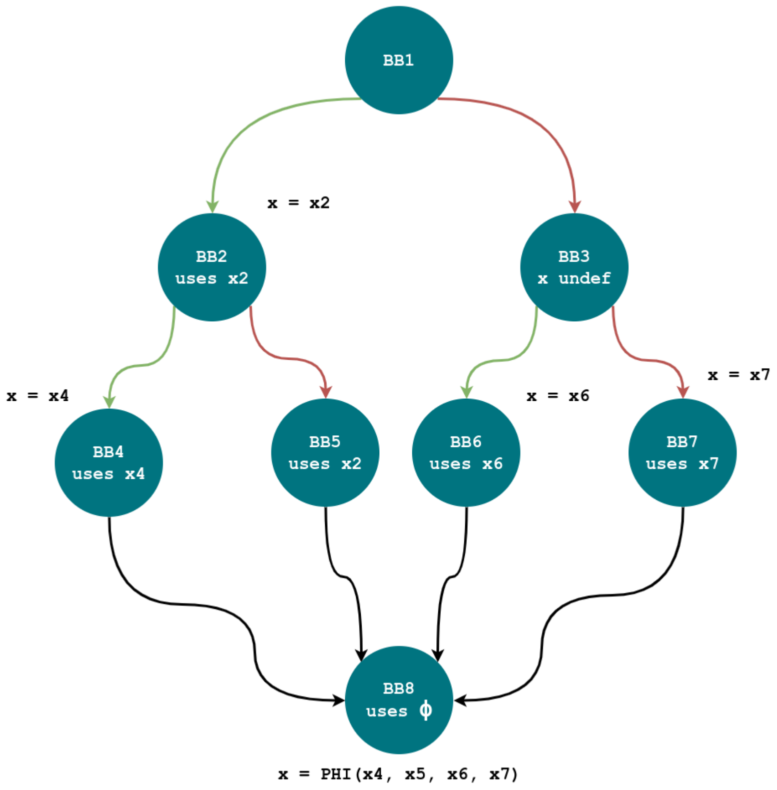


Figure 3.11: Example 2 of γ algorithm.

The γ tree is constructed as follows:

1. Start with c_1. When c_1 is true, expand using c_2. If c_2 is true, V_4 remains; if c_2 is false, V_5 remains.

2. When c_1 is false, expand using c_3. If c_3 is true, V_6 remains; if c_3 is false, V_7 remains.

The resulting γ function is:

$$\gamma(c_1, \gamma(c_2, V_4, V_5), \gamma(c_3, V_6, V_7)).$$

3.4 Implementation Details

One of the primary challenges in this work was integrating the requirements described in previous sections into the existing Dynamatic codebase. The implementation not only needed to ensure correctness but also had to maintain a high degree of readability and maintainability, so that later refactoring and corrections could require the least possible effort.

Another significant challenge arose from the nature of the algorithm discussed in earlier sections. The methodology describes a one-way transformation from the `cf` dialect to the `handshake` dialect. In essence, the algorithm applies transformations between every pair of producer and consumer. Consider a scenario where the entire transformation process is complete. Later, a pass introduces a new component in `BBx` that is connected to a component in `BBy`. This new connection must respect the *Fast Token Delivery* methodology, with the appropriate suppression and regeneration mechanism, to guarantee correctness. Therefore, the methodology should be adopted anywhere in the flow, possibly on single new subsections of the circuit (making sure not to add any unnecessary components).

Notice that, at this stage, it is not possible to get rid of the *cmerge network*, since it is still required to activate the group allocations for the LSQ. This problem will be solved in Chapter 4, with a circuit that interfaces with the LSQ while honoring the minimum necessary control flow decisions.

This section provides a detailed bottom-up explanation of the implementation structure, addressing these challenges.

3.4.1 Add Regeneration to a Pair of Producer and Consumer

The goal of regeneration, as expressed in Subsection 3.2.2, is to ensure that a value is available at each loop iteration for a given pair of producer and consumer. If multiple nested loops exist between the production and usage, the value must be regenerated at each loop level to maintain consistency. Figure 3.12 shows an example in which a value must be regenerated many times since the consumer is in a nested-loop structure.

In the context of MLIR compliance, a producer and consumer are not treated as *operations*. Instead, the methodology addresses each consumer and its associated operands. This approach accounts for scenarios where a single producer provides multiple values to the same consumer. Furthermore, a consumer might use the same value multiple times, reflecting the distinction between a *value* and a *use* in MLIR. The former is a value in the SSA format, while the latter is the specific usage of such a value.

Another consideration is that certain values might lack an explicit operation as their producer: this is the case for function block arguments, which serve as implicit producers.

The function `addRegenOperandConsumer` manages the insertion of regeneration `MUX`es. These `MUX`es are essential for maintaining the token count across loop iterations, thus all the pairs of producers-consumers should be checked to see if there is a loop the consumer is inside but the producer is not. However, within the circuit, the *cmerge network* already has a valid token count, thus it does not need any regeneration; moreover, Dynamatic models memory controllers as operations that are outside of any loop, so they do not undergo this process.

Several optimizations can enhance the regeneration process. For instance, a set of regeneration `MUX`es does not need to be created for each value individually. If a `MUX` is placed at a particular loop level for a value `x`, it can be reused multiple times within that loop. This reuse minimizes resource overhead while maintaining functionality.

3.4.2 Add Suppression to a Pair of Producer and Consumer

What was done for regeneration must also be implemented for the suppression mechanism. However, suppression introduces significantly more challenges due to the complex conditions that need to be generated and applied. As stated in the context of regeneration, suppression operates on a pair consisting of an operand and a consumer at a time, through the function `addSuppOperandConsumer`.

Certain situations must be excluded from the suppression mechanism to ensure proper functionality and avoid redundancy. In particular, all GSA gates, control merges, and other conditional branches should be skipped. While the exclusion of GSA gates and control merges is straightforward - as it was in the regeneration process - conditional branches require more explanation. Conditional branches are introduced into circuits only by the suppression mechanism itself, so suppressing something that has already been suppressed is not only unnecessary, but also invalidating the correctness.

Furthermore, if the producer and the consumer are within the same block, suppression is generally redundant. One exception exists: if the consumer is a `MUX`, suppression is necessary to prevent a value from being regenerated at the end of the loop execution (see Figure 3.12).

The relationships between a producer and a consumer can be categorized into four distinct scenarios:

1. **More producers than consumers**. When a value is generated inside a loop and used outside of it, suppression is required to discard all tokens except the final value produced in the last iteration. The branch condition in this case corresponds to the loop exit condition. The suppression mechanism connects the original operand to the *true* output of the conditional branch, ensuring that tokens are suppressed as long as the loop condition remains true.

2. **The producer is also the consumer**. This scenario involves a self-regenerating `MUX`, as mentioned in Subsection 3.4.1, where the producer regenerates its value. It is similar to the previous case, but the operand is connected to the *false* output of the conditional branch. In this configuration, tokens are suppressed only when the loop terminates, maintaining proper synchronization in the circuit.

3. **Backward edge**. This case occurs when there is an edge from `BBx` to `BBy`, where `BBy` dominates `BBx`. The necessity of suppression depends on whether the loop uses a `while` or `do...while` construct:

   - For a `while` loop, suppression is unnecessary because the loop header executes unconditionally every time the backward edge is traversed.
   - For a `do...while` loop, suppression is essential since the backward edge may not always execute. In this case, the value is connected to the *false* output of the conditional branch to ensure proper token flow management.

4. **Direct suppression**. When the producer and consumer are at the same loop level, direct suppression is applied using the algorithm described in Section 3.2. This

approach relies on a custom boolean logic library to handle the required conditions efficiently. Direct suppression is simpler than the other cases as it does not need to manage complex loop conditions or backward edges. An example of such method is shown in Figure 3.13.

3.4.3 GSA Transformation

The GSA transformation described in Section 3.3 is implemented through two key components: the *GSA Analysis Pass* and the `addGsaGates` function.

An analysis pass in MLIR is in charge of extracting and analyzing information about the IR without modifying it. They are in charge of keeping some information shared and consistent across multiple passes, so that consistency is maintained. For instance, a control dependency analysis can generate all the information related to the control dependency graph.

In the context of GSA, the analysis pass operates on the `cf` structure to extract GSA-related information without altering the IR. This information is then used to replace block arguments with `MUX`. A key advantage of this approach is that the extracted information remains reusable at later stages of the compilation process. The analysis pass is designed as a class capable of processing both the `cf` IR and `MERGE`s. Since a merge represents the instantiation of a ϕ function (sharing the same non-deterministic behavior), it can also be used to construct GSA gates.

Once the analysis has been completed and the required information is available, the `addGsaGates` function removes block arguments from each block. This function replaces them with either a single `MUX` (for μ gates) or a tree of `MUX`es (for γ gates), ensuring the IR adheres to the GSA representation.

3.4.4 Conversion Pass

The conversion pass utilizing the *Fast Token Delivery* methodology builds upon the previously described functionalities. The following sequence outlines the conversion process:

1. All block arguments are converted into GSA gates.
2. Memory controllers, either a standard *memory controller* or a *load-store queue*, are instantiated.
3. Constants and undefined values in the `arith` dialect are converted to their `handshake` equivalents.
4. The regeneration mechanism is applied to all producer-consumer pairs in the IR.
5. The suppression mechanism is applied to all producer-consumer pairs in the IR.
6. Individual operations are converted to their `handshake` equivalents.

This structured process ensures that the IR is fully transformed to leverage the fast token delivery methodology while maintaining consistency and correctness across all stages.

3.4.5 Peephole Optimizations

Although the flow described so far is capable of producing a correct circuit, it often introduces redundant components. These redundant components must be removed to minimize the resulting area while preserving both functionality and performance. This optimization is achieved through an additional transformation pass, called *Combine Steering Logic*.

The transformation pass utilizes `applyPatternsAndFoldGreedily` [33], a utility function designed to apply a set of rewrite patterns to the IR in a greedy manner. This utility iteratively traverses the operations in the IR, applying the provided patterns wherever they match and making direct transformations to the IR. After a pattern is applied, the utility revisits affected operations to determine if further patterns can be applied. This process continues until no more patterns remain applicable.

Below is a list of the patterns applied during this pass:

1. `RemoveSinkMuxes`. If the output of a `MUX` is not used, the `MUX` itself can be safely removed. This situation might seem counterintuitive—why would unused `MUX`es exist? The issue arises from how MLIR operates. When an operation is removed, it remains in the IR and can still be accessed through the API until the pass finishes. Consequently, a consumer may still appear in the analysis, even if it has already been deleted. To address this, the `MUX` is initially added but must be removed before execution continues.

2. `RemoveDoubleSinkBranches`. Similar to the case of `MUX`es, branches with both outputs unconnected can also be removed.

3. `CombineBranchesSameSign` and `CombineBranchesOppositeSign`. If two branches share the same data input and a *similar* condition input, one of them can be eliminated and replaced by the other. In this context, *similar* refers to conditions differing only by a *not* gate feeding the branch. By removing the *not* gate and swapping the branch outputs accordingly, the circuit remains functionally identical.

4. `CombineInits`. The only `MERGE` operations in the circuits are those created for `INIT` components, as illustrated in Figure 3.6. If multiple merges have identical inputs (e.g., a *false* constant and the same condition), one merge can be removed and replaced by the other.

5. `CombineMuxes`. `MUX`es used to regenerate the same value at the same loop level can be merged into a single `MUX`.

6. `RemoveNotCondition`. If the condition of a branch is fed by a *not* gate, the *not* gate can be removed. In this case, the branch outputs are swapped to preserve functionality.

These peephole optimizations are greedy, thus sub-optimal at the software level by construction. However, the resulting circuits are optimized in accordance with the expectations.

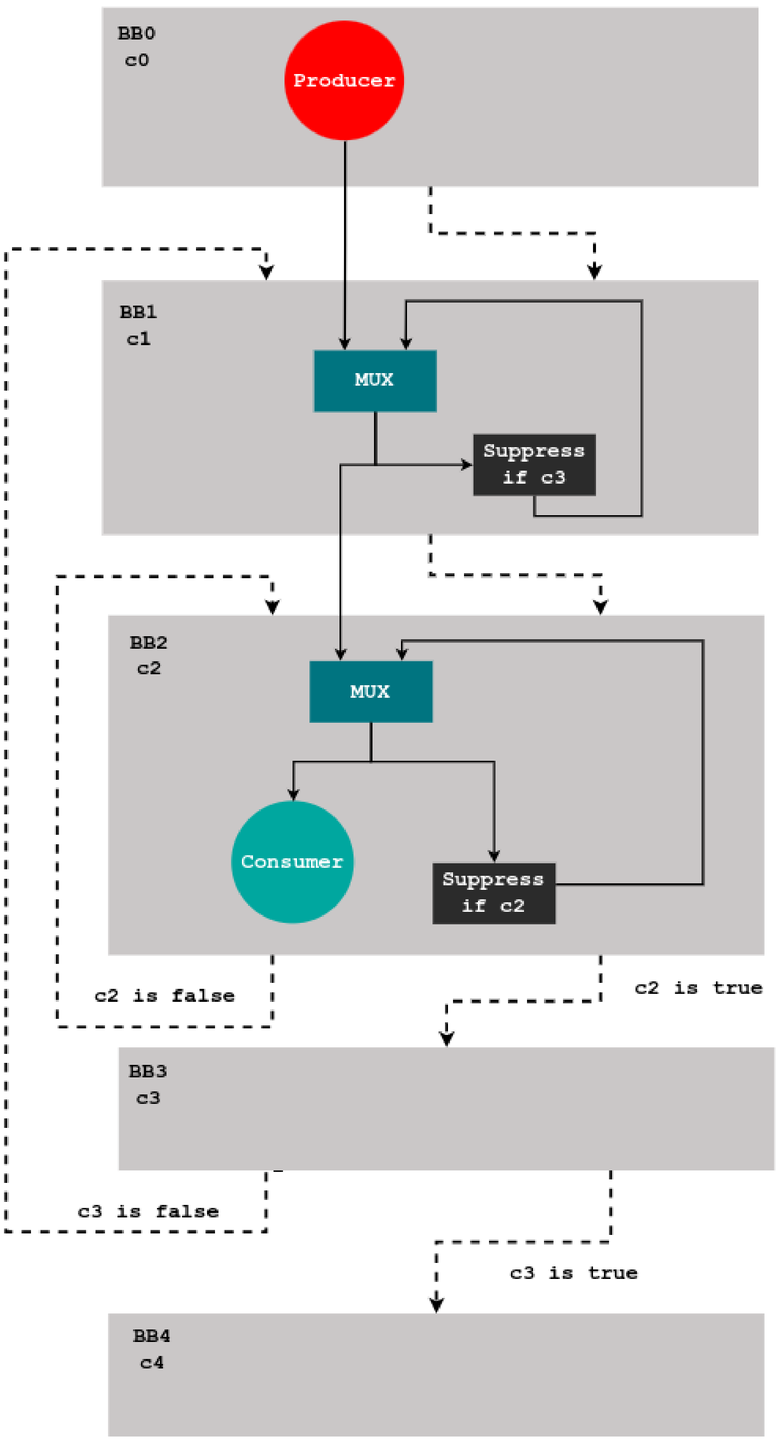


Figure 3.12: Example of a value regenerated at multiple levels due to some nested loops. Solid lines represent dataflow connections, while dashed lines represent control flow between BBs.

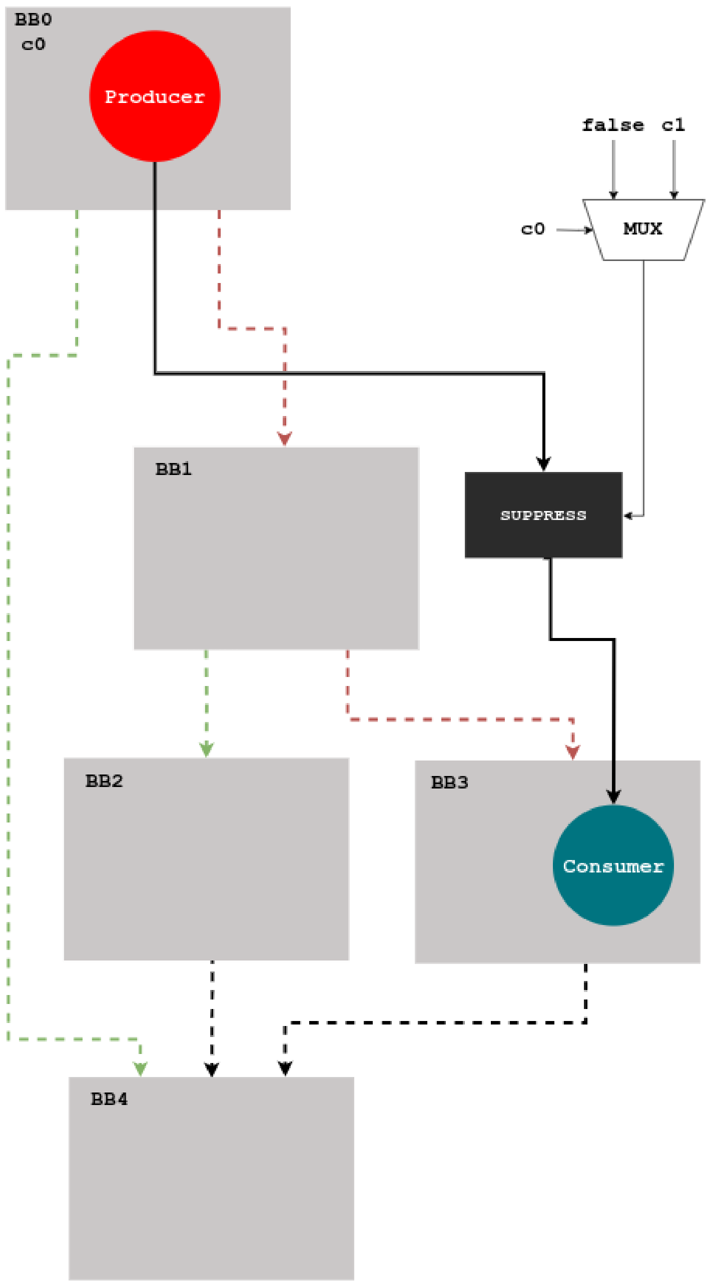


Figure 3.13: Example of direct suppression between producer and consumer.

Chapter 4

The *Straight To The Queue* Methodology

4.1 Memory Interconnection in Dynamatic

The discussion in Subsection 2.4.2 and Chapter 3 focuses on explicit data dependencies without addressing the issue of memory accesses. The memory-subsystem works the same way as explained in Subsection 2.4.4, with an LSQ to handle conflicting accesses and a sequential network to activate these accesses.

Since it is the core of the current chapter, an overview of memory integration is worthwhile.

Load and *store* operations are connected to specific memory regions; these regions consist of BRAM inside an FPGA, but more generally it is just some IO pins for the produced circuit (with data and address channel, some activation pins, and a *ready*/*valid* signals to guarantee the same handshake protocol everywhere). However, since usually multiple memory operations require the same memory region, a direct connection cannot be adopted; A *memory controller* is connected to $N \geq 0$ load operations and $M \geq 0$ store operations on the circuit side and interfaces with one memory region on the other side ($M + N > 0$). It handles circuit requests and, in the case of *load* operations, provides the corresponding data back.

In dataflow circuits, the execution order does not follow the original program sequence. This may lead to a scenario where a memory operation later in the code executes before an earlier one. Such reordering is acceptable as long as no *hazards* occur.

A common example of a hazard arises when one instruction performs a *store* to a memory location, and a successive *load* uses the same memory region: this is a typical *Read-After-Write* (RAW) scenario. If the *load* operation is issued before the *store* completes, the system may read outdated data, leading to incorrect results. An example of such an issue is shown in Listing 4.1.

Notice that such a situation can also be found in out-of-order CPUs, having memory instructions with some dependencies that force them to run in-order.

```
for (int i = 0; i < N; i++) {
    int val = A[i];
    if (val >= 0)
        tmp += val;
    A[i] = 0;
}
```

Listing 4.1: Kernel with memory dependencies that cannot be resolved without a load-store queue.

Several mechanisms exist to address such situations. For instance, in many cases relying on a static analysis can be enough to determine that a conflict cannot occur. For example, [7] proposes a static method to get rid of a memory dependency whenever there is also a data dependency involved (for instance, if the store operation from Listing 4.1 requires to use loaded data in line 2). However, static analysis has some obvious limitations, and a dynamic management is still inevitable.

To address this problem, [38] introduced the aforementioned LSQ - similar to the one used in modern CPUs, to guarantee that memory accesses are always handled in the correct order despite the issuing order. However, the LSQ in a CPU works by relying on the in-order instruction decode stage: dataflow circuits do not have an immediate way to obtain the original program order. For the way Dynamatic works, however, a couple of observations can be made:

- Within each BB of the CFG, the operations are sequential (due to the linear structure of the IR), and it is then possible at compile time to determine the order at which operations should be performed to guarantee correctness.
- Although the program order is not strictly followed in a dataflow circuits, all the BBs are activated in order according to the original control flow graph; for this reason, it is still possible to build on the fly the order of the operations, thanks to the *cmerge network*.

Putting together this two information, the LSQ is designed in a way that takes care automatically of the order of the memory operations; it is also in charge of comparing the address of a memory operation with all the received addresses of the operations to be run. If a conflict exists, then the memory operation needs to stall, otherwise it can safely run.

The approach instantiates one LSQ for each set of accesses, which might lead to a conflict. This choice is mostly related to the area occupation, since the LSQ represents a large part of the area of a dataflow circuit (up to 90%, according to [7]).

4.2 Motivations for a Faster Memory Allocation

This method guarantees a correct way to handle memory accesses, but it has a significant overhead when it comes to maintaining the control signals. A major performance advantage of *Fast Token Delivery* circuits, discussed in Chapter 3, arises from eliminating the relationship between the dataflow and the control flow graph. However, the *cmerge network*(that strictly mimics the sequential control flow) still exists to notify the LSQ of a BB

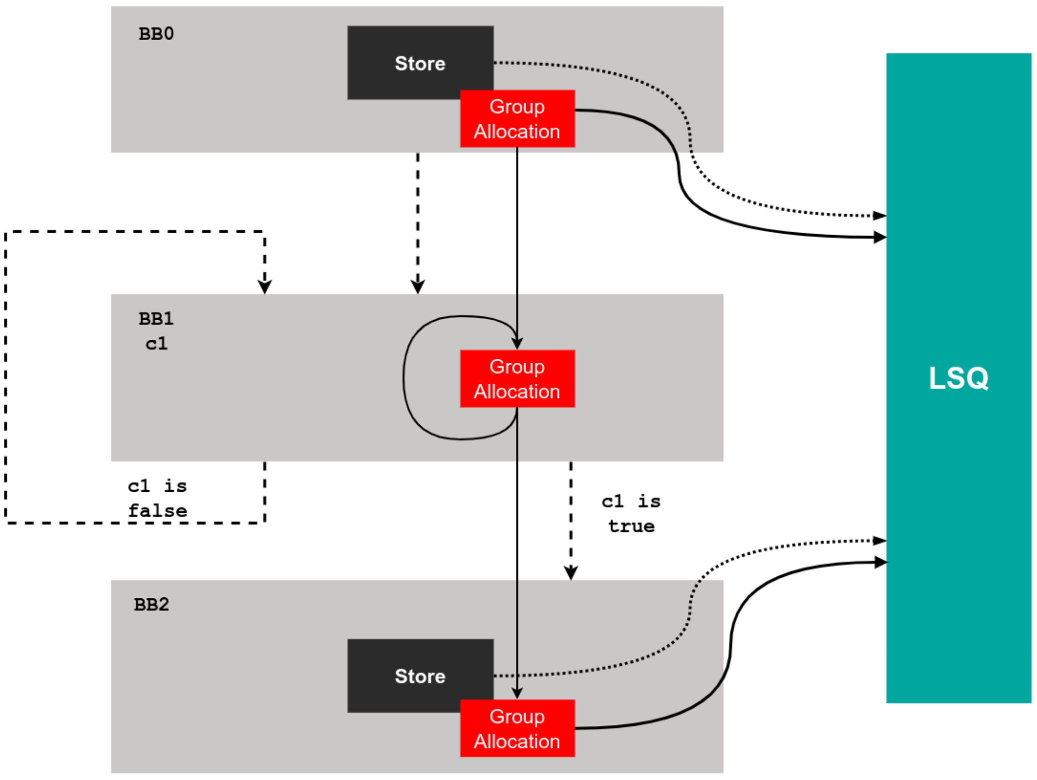


Figure 4.1: Interaction between BBs in the circuit and the LSQ; the *cmerge network* in red follows the program order, independently from performances.

allocation. How to get rid of such a network, and speed up the memory allocation? [12] has this objective.

It should be noted that such an approach is in charge of speeding up the interaction between the part of the circuit handling data and the LSQ; however, there might be other ways of handling conflicting accesses. Naively, one could design more complicated dataflow hardware within the circuit to detect address conflicts at runtime and to enforce the correct ordering of memory accesses only when the addresses collide (that is, obtaining the functionality of the load-store queue without using a load-store queue). Such a strategy is currently under study in the Dynamatic community, due to the area and timing overhead of an LSQ in the circuit.

4.3 Straight To The Queue Algorithm

4.3.1 A Minimal Group Allocation

The overall objective is to make sure that, for each memory operation connected to an LSQ, an *activation* token is generated, following the exact order of the original sequential program. This is the same objective of the original *cmerge network*, but avoiding to waste time in BBs that are not concerned with the LSQ.

A simple way to reuse the *Fast Token Delivery* approach is to manually insert a data-dependency between these operations, so that the allocation of a group of operations (corresponding to some operations within the same BBs) cannot happen before all the previous groups have been allocated. In the end, the objective is to obtain something similar to the *cmerge network* but in a minimized format, capturing the minimum necessary control flow decisions between memory dependencies.

Refer for instance to Figure 4.2; the first store operation from `BB0` must be allocated in the LSQ before the group for `BB2` is allocated; however, once the allocation of the former happens, it is guaranteed (because of dominator analysis) that `BB2` will run, so the activation can happen immediately, independently from the amount of times the loop `BB1` is executed. The original flow, on the contrary, relying on the *cmerge network*, required the loop to finish before allocating (Figure 4.1).

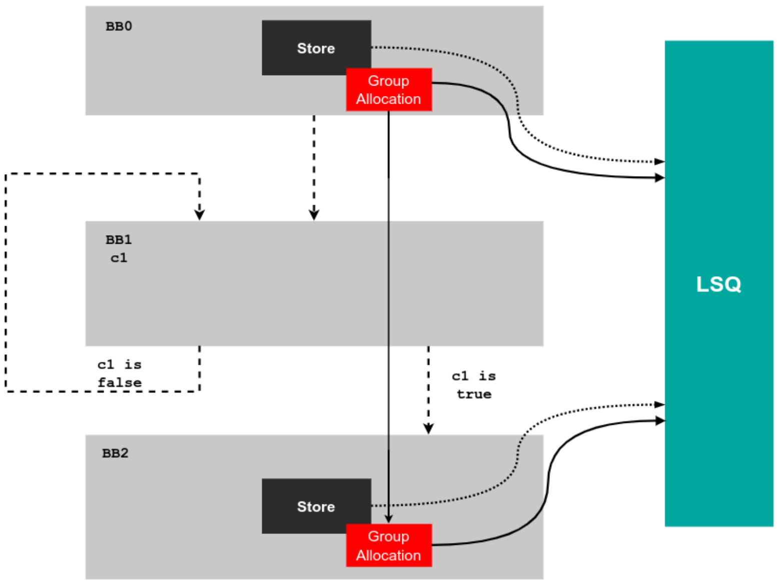


Figure 4.2: Example of simplified network for memory allocation.

This is the key element: guarantee a sequentialization of the accesses in a minimal format, without the need of relying on the entire CFG. Not only does this save components (thus area) but it also speeds up the execution of memory operations. For each BB connected to the same LSQ, a token needs to flow to guarantee correct allocation in the Queue.

4.3.2 Sequentialize Group Allocation

A specialized component called the sequentializer (`SEQ`) manages the distribution of these allocation tokens for a single BBs to the LSQ and subsequent circuit elements. The `SEQ` has multiple input ports which need to be joined before `SEQ` can fire. Each input token indicates that prior memory operations, which belong to the same in-order chain, have already been allocated. The way `SEQ`s from different BBs are connected reflects a subset of the original order of the program.

Once the `SEQ` fires, it sends one token to the LSQ, signaling that the set of memory accesses associated with its BB should be reserved in the LSQ. Another token is output to the downstream circuit elements to notify the allocation.

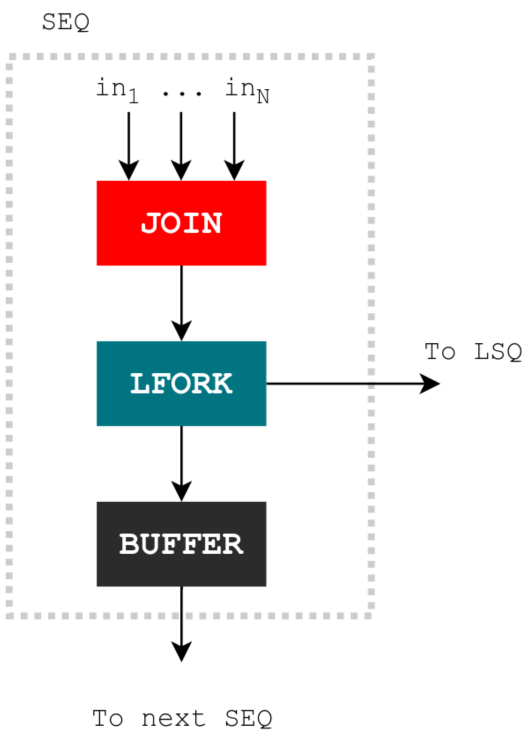


Figure 4.3: Exampe of `SEQ` circuit.

The internal design of the `SEQ` module utilizes three dataflow components (Subsection 2.4.2), as shown in Figure 4.3.

- A `JOIN` is used to merge incoming tokens from all predecessors. This step confirms that all branches responsible for feeding the `SEQ` have delivered a token, so that the allocations for the preceding groups have been done.
- A *lazy fork* (`LFORK`) replicates the token to two outputs: one directed to the LSQ and the other to any successor `SEQ`s. The `LFORK` distributes the incoming token only when all the successors are *ready* at the same time; this makes sure that the next successive `SEQ` is activated only after the memory allocation has been performed.
- A `BUFFER` introduces a single-cycle delay on the path to successor `SEQ`s. This is necessary to guarantee a delay between the allocation of two groups to the same

LSQ.

If the LSQ notifies (through backpressure) that it cannot accept any additional requests, the `LFORK` stalls the mechanism until it becomes ready again. This system stops successive allocations from bypassing earlier allocations, thus preserving the order seen in the original code.

4.3.3 Constructing the Allocation Network

Previously, the allocation into the LSQ had the granularity of the BB. Since the final objective is to get rid as much as possible of this concept, all the operations of an LSQ in the same BB are collected into a *group*.

Before running the algorithm, the following setup is available: there are some memory operations connected to some LSQ; S_i is the set of operations that are connected to the LSQ i. It is reasonable to say that S_i is already in a minimal format, without the possibility of further reduction through static analysis; otherwise, extra analysis can be placed before this step in the compiler flow.

Grouping and Graph Representation

Once the memory operations are collected according to their original BBs, we end up with a set of *groups*, S'_i. For each group in this set, a `SEQ` element needs to be instantiated. Notice that the grouping terminology is just a way to detach from the BB concepts; in reality, each `SEQ` has the same purpose of a `CMERGE` in the *cmerge network*.

A directed graph G_i is constructed where each node is an element of S'_i.

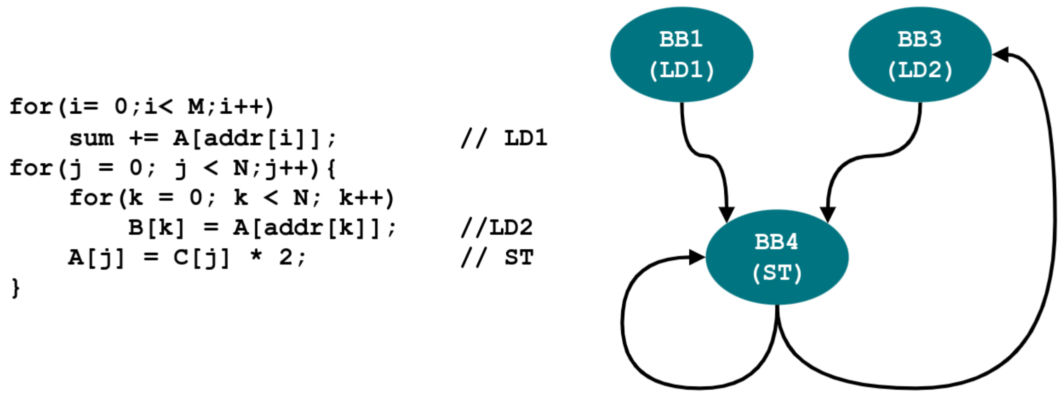
An edge from node u to node v in G_i highlights a potential memory dependency from any operation within the BB represented by u to any operation in the BB represented by v. Such edges indicate that v must wait for u to allocate its memory operations before v does: the `SEQ` of v will wait for the token of u to be released. Figure 4.4 shows a kernel and its corresponding graph. It is worth remembering that there is no hazard between two load operations. This graph reflects the fact that:

1. `ST` cannot be allocated before both the load operations are performed; however, after the first time it runs, the `ST` is also dependent on itself, since there might be a conflict too.
2. `LD1` has no incoming dependencies, so it can run whenever it is ready.
3. `LD2` can initially run whenever it is ready, but later it needs to wait for `ST`.

In the same example, the LSQ is required only for the memory region named `A`; `addr` and `C` are read-only, while `B` is guaranteed to have distinct accesses for each operation.

Inserting `SEQ`s

Out of this graph, it is possible to insert and connect directly the sequentizers. Each BB with a corresponding node in G has a `LFORK` fed by a `JOIN`. Such a component is connected to all the predecessors of the node in the graph.



```
for(i= 0;i< M;i++)
    sum += A[addr[i]];          // LD1
for(j = 0; j < N;j++){
    for(k = 0; k < N; k++)
        B[k] = A[addr[k]];      //LD2
    A[j] = C[j] * 2;            // ST
}
```

Figure 4.4: Example of a memory dependence graph for memory allocation.

Some graph nodes may lack any predecessors, indicating that no potential dependency precedes their operations. In this scenario, the *start* signal of the kernel is adopted for activation. This ensures that at least one token will flow into the node's SEQ, triggering allocation of that group of memory operations. On the other hand, some nodes have no successors, and in those cases, the SEQ output can terminate in a SINK, signifying the end of a dependency chain.

While this is a straightforward implementation of what was described so far, there are some clear problems when normal CFGs are used. Let's consider the CFG in Figure 4.5, in which dependent memory operations are shown in their corresponding BBs. LD can only run when either ST1 or ST2 are done. However, only one of the two operations will run, since BB2 and BB3 are mutually exclusive. If both the LFORKs of the two BBs are connected to the JOIN of BB4, then it will never fire the activation token, since one token will always be missing.

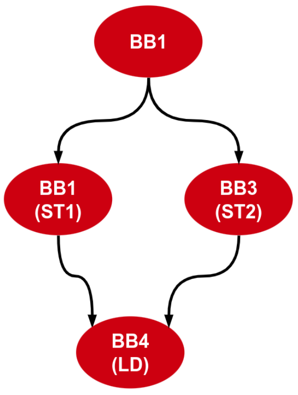


Figure 4.5: Example of a CFG with alternative memory operations.

4.3.4 Handling Alternative Incoming Activation Tokens

The main problem is that, for an IR with an underlying CFG structure, variables are supposed to be in SSA format. This means that, when alternative inputs are available due to alternative predecessors in the CFG, a ϕ node is supposed to exist.

As this is true for normal variables in the code, this should be true for the activations tokens as well. Consider again the example if Figure 4.5. Instead of the two sequentializers in BB2 and BB3 to feed the one in BB4, a ϕ node should exist which merges them accordingly to the control flow decision.

This scenario is to be handled also for Figure 4.4. Considering also the *start* signal, the circuit in Figure 4.6 will be obtained. This shows that the first allocation of BB4 is made by the *start* signal, while all the subsequent ones depend on ST itself; same goes for LD2.

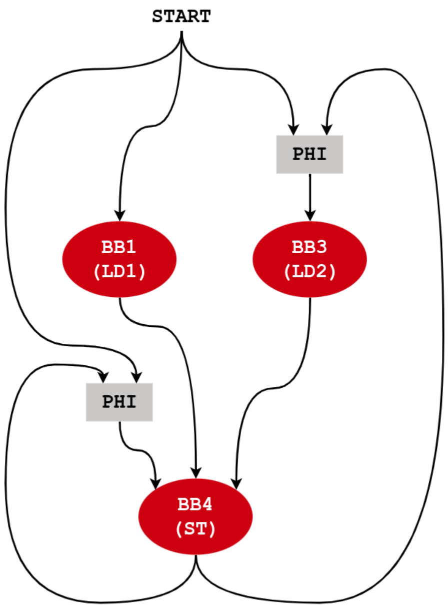


Figure 4.6: Example of a memory allocation graph with the correct ϕ nodes to guarantee correctness.

4.4 The PHI Insertion Mechanism

As described in [12], the implementation of such allocation techniques relies on a standard SSA mechanism to insert ϕ nodes.

In the context of MLIR with the `handshake` dialect, there is no optimal automated way to insert these nodes. This process must be handled manually, referring to standard algorithms.

4.4.1 Problem Statement

A variable is defined multiple times across N different nodes: v_1, v_2, ..., v_N. In this context, v refers to the variable that generates the allocation token upon which the sequentializer depends. The problem is to determine which value of the variable to be used in each BB, possibly merge some of them with ϕ nodes.

4.4.2 The Algorithm

The following steps outline the process:

1. Collect all the operations which define the value and the corresponding BBs. In general, a value can be defined multiple times within the same BB.

2. For each BB, sort the operations according to the dominance information and keep only the last one. From a linear perspective, the last operation which defines a value within the BB is the one determining the value to be used afterwards.

3. Use the Cryton algorithm [26] (described later) to identify the BBs where ϕ nodes need to be inserted. In some cases, ϕ nodes may not be required, depending on the dominance tree.

4. Each ϕ node requires values from every predecessor of that BB. These values can originate from a definition within the predecessor, a ϕ from the predecessor itself or a subsequent predecessor of a predecessor.

5. Instantiate the ϕ node in each BB with the appropriate connections.

The Cytron algorithm, as presented in [26], is shown in Listing 4.2.

It takes as input a region with an underlying CFG structure and a set of values defined within specific blocks (`inputBlocks`). The first step involves computing the *dominance frontier* [39] for each BB. Given a node d in a CFG, its *dominance frontier* is defined as the set of nodes n such that:

$$d \text{ dominates a predecessor of } n \text{ but does not strictly dominate } n.$$

The Cytron algorithm uses a method to find the dominance frontier of each node and three data structures:

- `w`: A set of nodes that still need to be analyzed.
- `hasAlready`: A map indicating whether a block has already been added to the list of nodes requiring a ϕ node.
- `work`: A map of nodes that have already been processed.

```
/// Run the Cryton algorithm to determine, given a set of values,
    in which blocks
/// a merge is needed to ensure those values are correctly
    propagated.
static DenseSet<Block *>
runCrytonAlgorithm(Region &funcRegion, DenseMap<Block *, Value> &
    inputBlocks) {
  // Get dominance frontier
  auto dominanceFrontier = getDominanceFrontier(funcRegion);

  // Temporary data structures to run the Cryton algorithm for phi
     positioning
  DenseMap<Block *, bool> work;
  DenseMap<Block *, bool> hasAlready;
  SmallVector<Block *> w;

  DenseSet<Block *> result;

  // Initialize data structures
  for (auto &bb : funcRegion.getBlocks()) {
    work.insert({&bb, false});
    hasAlready.insert({&bb, false});
  }

  for (auto &[bb, val] : inputBlocks)
    w.push_back(bb), work[bb] = true;

  // Process the list until `w` is empty
  while (!w.empty()) {

    // Pop the top of `w`
    auto *x = w.back();
    w.pop_back();

    // Get the dominance frontier of `x`
    auto xFrontier = dominanceFrontier[x];

    // Process each element in the frontier
    for (auto &y : xFrontier) {

      // Add the block in the dominance frontier to the result list
         .
      // If it was not analyzed yet, also add it to `w`
      if (!hasAlready[y]) {
        result.insert(y);
        hasAlready[y] = true;
        if (!work[y])
          work[y] = true, w.push_back(y);
      }
    }
  }

  return result;
}
```

Listing 4.2: Cryton Algorithm in MLIR.

Notice that the whole work of this thesis focuses on GSA rather than SSA. Once the ϕ information is obtained, the method from Section 3.3 can be applied to convert each of these nodes into a μ or a γ.

4.5 Implementation Details

The methodology described thus far is implemented in Dynamatic through an additional pass that operates on the `handshake` dialect after its conversion from the `cf` dialect. Since the components inserted during this stage must still undergo the *Fast Token Delivery* methodology (specifically, the insertion of the suppression and regeneration mechanisms) it was essential to design the algorithm in Chapter 3 in a modular and reusable way, independent of the conversion pass.

This pass is optional. During the conversion process, the *cmerge network* is instantiated and used. Later, this network can optionally be disconnected and replaced with sequentializers. At this point, the original *cmerge network* can be removed, as its purpose is terminated.

For the pass to function correctly, an underlying CFG structure must exist in the IR. However, the `handshake` dialect is represented as a *graph region* with no BBs; everything is flattened, as would be expected from the final circuit. To meet this requirement, the following steps are employed:

1. Before the conversion from `cf` is finalized, each operation in the IR is annotated with metadata identifying the BB to which it belongs.

2. The MLIR operation of the function is annotated with information about all edges between BBs;

3. When the CFG is required, BBs are re-constructed and utilized using the edge annotations and the block-specific metadata in each node. Although this reintroduction of BBs deviates somewhat from standard MLIR practices, it functions as intended.

4. The resulting IR from the pass cannot include BBs due to the graph region requirements of the `handshake` dialect. Therefore, the recreated blocks are removed just before the pass concludes.

This mechanism is encapsulated in a new class called `CFGAnnotation`.

Chapter 5

Experimental Results

This chapter highlights the results obtained from the techniques adopted from [12] and [10], and whose implementation was described in Chapter 3 and Chapter 4.

5.1 Experimental Setup

5.1.1 Methodology

To perform a fair comparison of the baseline —referred to as *legacy*, commit `637df68` from [16] — and the implemented work — referred to as *FTD* — the following setup has been adopted.

For both versions, all available optimizations have been turned on; in particular, they both benefit from the reduction of the size of constants and operations, as described in Subsection 2.4.4. The two resulting circuits are tested using the testing infrastructure that is already integrated in the compiler. This consists of running the original C program, then running an RTL simulation, and checking that the content of the memories and the returned value of the function are identical. This methodology ensures the circuit is always functionally correct.

The RTL simulations are run using Modelsim 20.1. Such simulations allow us to obtain the simulation time with a clock period set at $4ns$. From this, it is straightforward to obtain the number of clock cycles for the simulation.

Results cannot be compared without considering area and critical path. To obtain these metrics, both *legacy* and *FTD* are synthesized using Vivado 22.1, targeting a Kintex-7 Xilinx FPGA with a constraint on the critical path of $4ns$. The resource usage from Vivado is obtained after place and route, and it consists of LUTs, FFs, and Slices.

Only the VHDL modules have been used, without relying on Verilog, whose implementation in Dynamatic still lacks many FPU components. `FTD` version adopts both *Fast Token Delivery* and *Straight To The Queue*.

One could argue that testing the two algorithms at the same time is not a reliable measure of performance. However, two points should be taken into account:

1. *Straight To The Queue* is only compliant with *Fast Token Delivery*, and the algorithm cannot run without it.

2. Without *Straight To The Queue*, *Fast Token Delivery* still relies on the *cmerge* network to allocate memory groups to the LSQ.

   This means having an implicit sequential execution of the BBs, which undermines the benefits of *FTD*.

5.1.2 Heuristic Buffering Algorithm

As described in the introduction of this work, when discussing the performance of Dynamatic, it is not possible to obtain reliable circuits in terms of throughput without a clever buffering algorithm. [9] and [11] are already implemented in the flow; by relying on an MILP formulation of the problem, they both allow for optimal buffering of the standard Dynamatic.

However, neither algorithm is compliant with *Fast Token Delivery*, due to the naive way they are implemented. The main assumption of these works is that a connection between two components can exist only if the components are located in two adjacent BBs - with respect to the CFG of the program. This implicit coupling between dataflow and control-flow is disrupted by *Fast Token Delivery*, which aims to build connections as directly as possible.

Since this work was not about fully integrating *Fast Token Delivery* with all the previous work on Dynamatic, these techniques could not be used. However, the so-called *naïve buffering*, which consists of placing a fixed number of transparent buffers blindly almost everywhere, could lead to enormous inefficiencies. The decision was then to build a heuristic. To ensure fairness in the comparison, such a heuristic has been applied to both *legacy* and *FTD*.

To guarantee that each combinational cycle is broken, an opaque buffer (introducing one clock cycle of latency) is placed after each `MUX`. Then, for each output channel of a *fork* component, five transparent buffers are inserted. The number adopted is completely arbitrary, and no other value was tried in the process.

Not all the fork outputs require a transparent FIFO. The heuristic first simulates with all the buffers present, then tries to remove each buffer one by one, launching the simulation again. If the removal of the buffer does not increase the runtime of the kernel, it means that the buffer was not affecting throughput and can be safely removed; otherwise, the transparent buffer is kept. Figure 5.1 depicts the process.

As Section 5.3 will show, this process is sometimes sub-optimal, both for the resulting critical path and the area occupancy. For instance, if only a one-element FIFO was necessary over a channel, this method would keep the five buffers anyway. Many improvements can be considered; however, running so many simulations is time-consuming even for high-end server hardware, so these results are considered acceptable.

5.2 Functional Results

Dynamatic has 88 integration tests, consisting of various kernels used to verify different system functionalities and trigger potential corner cases. Of these tests, 14 do not work

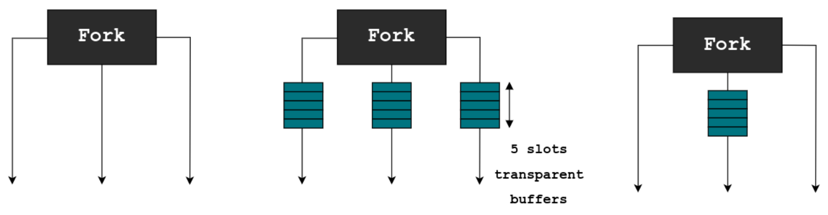


Figure 5.1: Exemplification of the heuristic buffering method: first, 5 transparent buffers are inserted at the output of each fork; then, only the buffers that affect throughput are kept.

on the baseline either (as of December 2024). This is due to multiple reasons, including failures in the LSQ functionalities and the lack of VHDL implementations for long-latency operations required by the compiler (e.g., `sqrt`, `cos`). For this reason, only the remaining 74 tests are considered.

When running the compiler with the *Fast Token Delivery*, all 74 tests pass successfully. This is a promising result, highlighting the reliability of the implementation and its ability to handle a variety of scenarios present in the original CFG. This result is the consequence of multiple ad-hoc tests, addressing one by one many situations; however, a comprehensive unit test-suite is not available.

The implementation of *Straight To The Queue* fails in four tests: `insertion_sort`, `gemver`, `gemver_float` (these two tests share the same CFG but use different types of arithmetic units, making it reasonable that both fail), and `test_memory_5`. In all these cases, the simulation stalls without terminating.

Particular attention was given to `gemver`. An analysis of the waveforms indicates that, as expected, *SQ* can result in multiple requests for group allocation to the LSQ within the same clock cycle. This situation occurs, for instance, with the graph allocation (see Section 4.3.3) depicted in Figure 5.2, which mirrors the behavior of one of the LSQs in `gemver`.

In this case, the group from `BB1` requires a store operation, which conflicts with the two load operations in `BB2` and `BB3`. This means that the latter two operations cannot start until all groups for `BB1` have been allocated. However, once `BB1` completes all the iterations, `LD1` and `LD2` are no longer in conflict and can theoretically run in parallel. When the LSQ was designed in [34], it assumed that BBs would execute sequentially in program order, allowing only one group allocation per clock cycle. This limitation compromises generality.

However, it is important to note that this issue falls outside the scope of the *Straight to the Queue* algorithm, which is only responsible for delivering tokens to allocate groups to the LSQ. Thus, the LSQ should be improved to optimize this use case.

If no modifications are made to the LSQ, alternative solutions can be considered:

1. **Introduce an arbiter for group allocations in front of the LSQ**. This approach is inefficient in terms of area but allows a dynamic strategy to mitigate the issue. Such arbiter receives all the allocation requests; if there is only one request at a given clock cycle, this can bypass the arbiter and go directly to the LSQ; otherwise, an arbitration over multiple requests (for instance, priority-based) can be done.

2. **Sequentialize group allocations at the graph level**. No issue arises if there are no two group allocations in the same clock cycle. This can be ensured by introducing *dummy* dependencies. For example, in Figure 5.2, adding a dependency between `BB2` and `BB3` would force `BB3` to execute only after `BB2` completes. This option requires a formal definition of the characteristics of the allocation graph related to a LSQ so that it is safe; no effort has been put in this direction.

3. **Halt the algorithm when a stalling risk is detected**. This scenario has not been formally defined. However, the idea would be to identify graph conditions that always lead to a stall and halt the algorithm beforehand, maintaining the *cmerge network* in charge of allocating the groups.

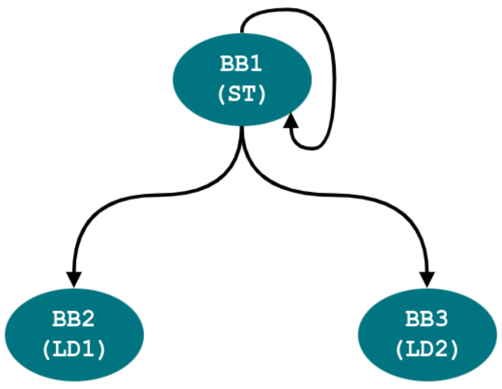


Figure 5.2: Example of a Memory Allocation Graph that causes LSQ failure.

Overall, the functional results demonstrate success, showing that *Fast Token Delivery* can be adopted for any kernel, while *Straight To The Queue* is safe in many situations.

5.3 Performance Results

A comprehensive scatter plot showing benchmark results using the setup from Section 5.1 can be found in Figure 5.4 (comparing timing and LUT usage) and Figure 5.3 (comparing timing and FF usage). A more general result is shown in Figure 5.5, where area is represented using *slices* in the FPGA. All numerical results are summarized in Table 5.1

(number of clock cycles and clock period), Table 5.2 (execution time), Table 5.3 (flip-flop usage), and Table 5.4 (LUT usage).

Execution time is estimated by multiplying the number of clock cycles obtained from the simulation by the critical path after place and route, since that is a lower bound for the clock period in the FPGA.

Here is a short description of each adopted kernel. Some of the tests are taken from *PolyBench* [40].

- `binary_search`: implements a binary search of a number over an array;
- `fir`: computes the *finite input response* between two discrete signals;
- `gcd`: compute the greatest common denominator between two integers using Stein's algorithm;
- `get_tanh`: implements a kernel with a long latency loop carried dependency in the loop body;
- `jacobi_1d`: from the *PolyBench* test-suite, it is the 1-D Jacobi stencil computation;
- `kernel_2mm`: from the *PolyBench* test-suite, it performs a 2 matrix multiplication $alpha \cdot A \cdot B \cdot C + beta \cdot D$;
- `kernel_3mm`: from the *PolyBench* test-suite, it performs a 3 matrix multiplication $(A \cdot B) \cdot (C \cdot D)$;
- `matvec`: multiplication between a matrix and a vector;
- `sobel`: computes the *sobel filter* between an input and two 3×3 kernels passed as input;
- `spmv`: implements a sparse-matrix multiplication by a dense-vector;
- `atax`: from the *PolyBench* test-suite, it implements a sequence of loops both memory dependencies and loop-carried dependencies;
- `bicg`: implements the BiConjugate Gradient STABilized method method;
- `stenci_2d`: it implements a grid-based computation between an original input matrix and a 3×3 kernel.

5.3.1 Timing Results

When discussing timing, it is possible to separately consider the simulation time (number of clock cycles, shown in columns 2, 3, and 4 of Table 5.1) and the critical path (columns 5 and 6 of the same table).

FTD improves the simulation time for all benchmarks, with an average reduction factor of 0.67. This indicates that simplifying the dataflow structure by decoupling it from the control flow is beneficial.

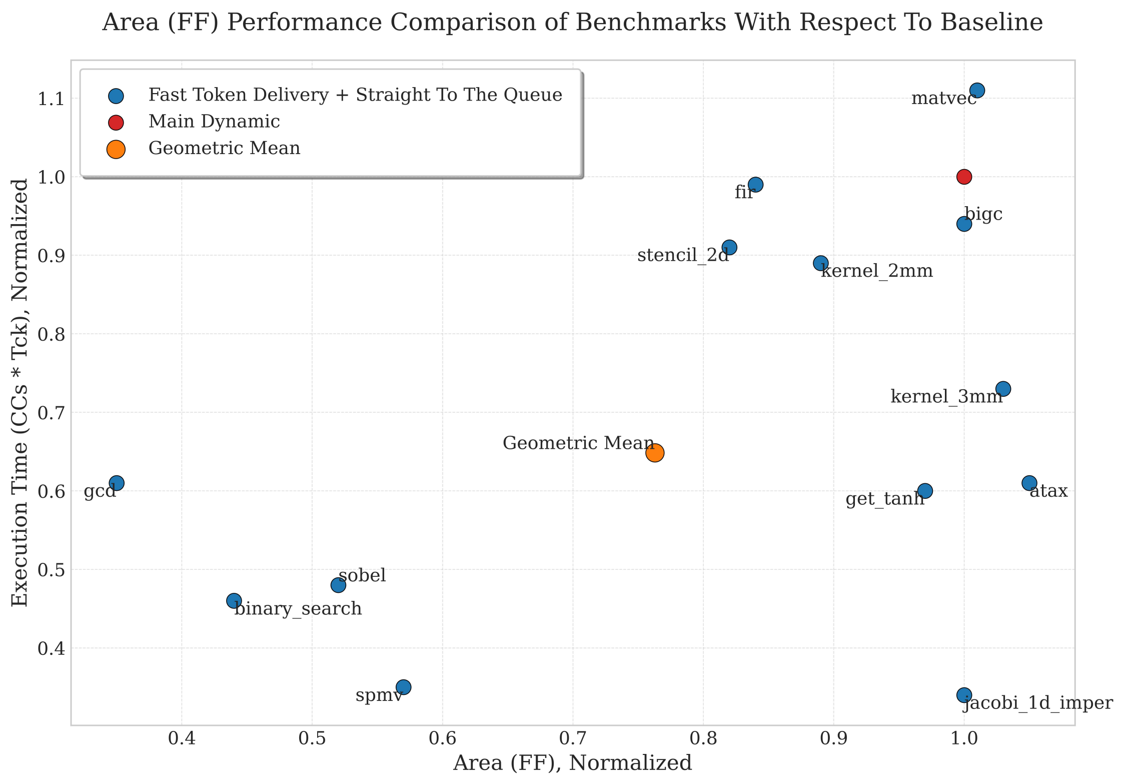


Figure 5.3: Plot showing execution time versus flip flop usage of *FTD* compared to the baseline.

`fir` is the simplest kernel in Dynamatic's test suite, where there is little to simplify since the circuit generated by *legacy* is nearly equivalent to that produced by *FTD*. Consequently, the number of clock cycles remains unchanged. Conversely, `spmv` and `binary_search`, which have complex and redundant *legacy* implementations, benefit significantly from the optimization. In Figure 5.5, these kernels are located in the bottom-left portion of the plot.

It is important to note that the number of clock cycles alone does not provide a complete insight into kernel execution time, as it also depends on the critical path after place and route. For this reason, the last columns of Table 5.1 present this metric. The results indicate that all kernels have a worse critical path, with the average ratio between *FTD* and *legacy* being 1.06. The minimum ratio is 0.99 (for `fir` and `bicg`), while the maximum is 1.22 (for `matvec`).

These results align with the conclusions from [10]. The reason might lie in the buffering algorithm from Subsection 5.1.2, which is not designed to optimize the critical path. However, since the same algorithm is applied to both *FTD* and *legacy* benchmarks, this is not a distinguishing factor. As it will be discussed in Subsection 5.3.2, *FTD* circuits typically contain more buffers than *legacy* circuits after the heuristic. Since transparent buffers do not introduce latency and thus do not break the critical path, their presence plausibly results in longer critical paths.

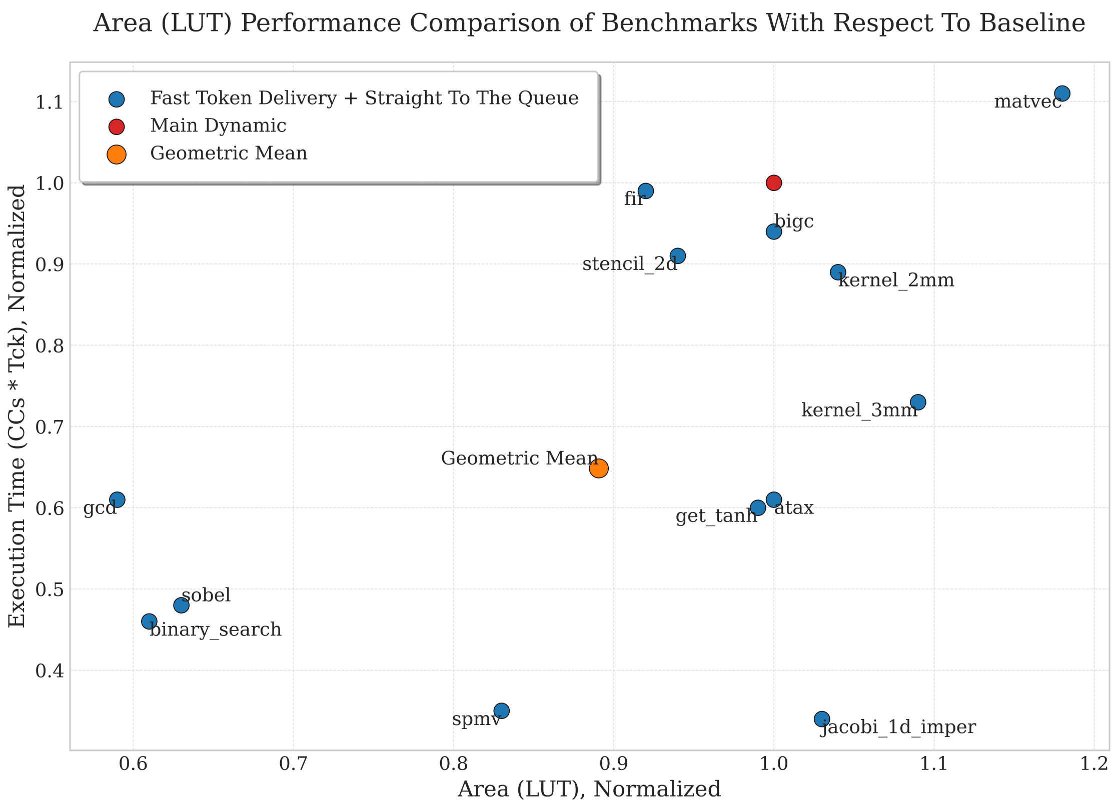


Figure 5.4: Plot showing execution time versus LUT usage of *FTD* compared to the baseline.

Ultimately, an estimate of execution time can be obtained by multiplying, for each kernel, the number of clock cycles by the critical path. These values are reported in Table 5.2, and their relationships can be observed on the vertical axis of Figure 5.4, Figure 5.5, and Figure 5.3.

With an average execution time ratio of approximately 0.7, the algorithm demonstrates the potential for timing improvements. Although the reduction in clock cycles is numerically greater, the increase in the critical path affects the overall results. Despite this, `matvec` is the only kernel with a worsened execution time. This can be considered a positive outcome, since this result is only due to the worst critical path. These results serve as experimental validation that *Fast Token Delivery* is effectively a minimal version of the simple *legacy* approach, generally leading to better results.

Potential strategies for further improvement are discussed in Section 5.4.

5.3.2 Area Results

The area results from this work are presented in Table 5.3 and Table 5.4.

On average, improvements are observed in both LUT and FF usage: LUTs are reduced by a factor of 0.91, and FFs by 0.81. This is coherent with the intuition according to which

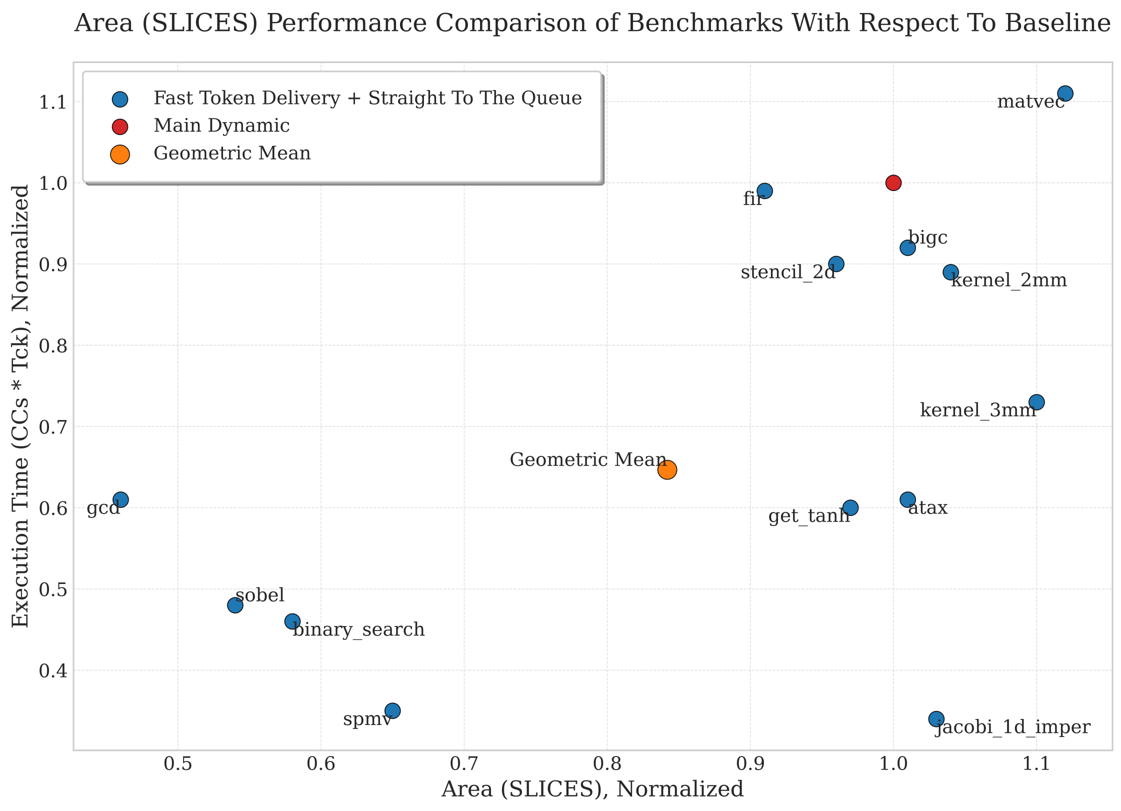


Figure 5.5: Plot showing execution time versus Slice usage of *FTD* compared to the baseline.

Fast Token Delivery removes steering components in the circuit, without introducing anything spurious. Rather than analyzing these elements separately, slices are used as a reference metric (considering both FFs and LUTs). In this case, the average improvement is 0.87, as shown in Figure 5.5.

While many benchmarks exhibit reduced area (some achieving a 0.5 factor improvement, such as `gcd` and `sobel`), others show an increase (right side of the plot). However, the increase is often minimal, with the worst case being `matvec`, which has an area factor of 1.12.

What causes this behavior?

As briefly mentioned in the previous section, the heuristic buffering algorithm introduces more transparent buffers in *FTD* circuits compared to *legacy*. To focus on the impact of transparent buffers, the worst-performing kernels have also been synthesized with no transparent buffers. While this scenario is suboptimal in terms of throughput (while leading to functional circuits anyways), it helps isolate the effect of transparent buffers on FPGA area. The results are summarized in Table 5.5.

The table clearly shows that *FTD* also achieves better area utilization (or exhibits only negligible increases) compared to *legacy* in all cases. This further supports the conclusion that the algorithm effectively reduces the number of components required for

Kernel	**Legacy CC**	**FTD CC**	**Ratio CC**	**Legacy CP (ns)**	**FTD CP (ns)**
binary_search	371	162	0.44	3.995	4.173
fir	1015	1014	1.00	3.728	3.679
gcd	137	83	0.61	4.590	4.631
get_tanh	7333	4355	0.59	7.552	7.680
jacobi_1d	486	165	0.34	7.153	7.182
kernel_2mm	1945	1704	0.88	6.471	6.601
kernel_3mm	2562	1925	0.75	6.540	6.361
matvec	456	415	0.91	3.342	4.089
sobel	5351	2165	0.40	4.779	5.728
spmv	106	34	0.32	3.949	4.287
atax	1682	1059	0.63	7.841	7.546
bigc	318	301	0.95	6.053	6.004
stencil_2d	496	428	0.86	3.751	3.977

Table 5.1: Clock cycles during simulation and clock period for *Legacy* and *FTD*.

Kernel name	**Legacy Time (μs)**	**FTD Time (μs)**	**Ratio Time**	**Improv. (%)**
binary_search	1.48	0.68	0.46	54.39
fir	3.78	3.73	0.99	1.41
gcd	0.63	0.38	0.61	38.87
get_tanh	55.38	33.45	0.60	39.60
jacobi_1d	3.48	1.19	0.34	65.91
kernel_2mm	12.59	11.25	0.89	10.63
kernel_3mm	16.76	12.24	0.73	26.92
matvec	1.52	1.70	1.11	-11.35
sobel	25.57	12.40	0.48	51.51
spmv	0.42	0.15	0.35	65.18
atax	13.19	7.99	0.61	39.41
bigc	1.92	1.81	0.94	6.11
stencil_2d	1.86	1.70	0.91	8.51

Table 5.2: Execution time for *Legacy* and *FTD*.

functional dataflow circuits.

Kernel name	**Legacy FFs**	**FTD FFs**	**Ratio FFs**	**REG Improv. (%)**
binary_search	2035	900	0.44	55.77
fir	449	378	0.84	15.81
gcd	2815	992	0.35	64.76
get_tanh	3374	3271	0.97	3.05
jacobi_1d	3783	3796	1.00	-0.34
kernel_2mm	6419	5745	0.89	10.50
kernel_3mm	7062	7297	1.03	-3.33
matvec	608	612	1.01	-0.66
sobel	3992	2082	0.52	47.85
spmv	2263	1293	0.57	42.86
atax	5041	5282	1.05	-4.78
bigc	3987	3994	1.00	-0.18
stencil_2d	1042	853	0.82	18.14

Table 5.3: Flip Flop Usage for FTD and Legacy.

Kernel name	**Legacy LUTs**	**FTD LUTs**	**Ratio LUTs**	**LUT Improv. (%)**
binary_search	1893	1164	0.61	38.51
fir	413	378	0.92	8.47
gcd	2545	1512	0.59	40.59
get_tanh	10600	10517	0.99	0.78
jacobi_1d	16750	17305	1.03	-3.31
kernel_2mm	20440	21287	1.04	-4.14
kernel_3mm	26421	28764	1.09	-8.87
matvec	581	683	1.18	-17.56
sobel	3659	2298	0.63	37.20
spmv	1659	1372	0.83	17.30
atax	18767	18822	1.00	-0.29
bigc	16655	16724	1.00	-0.41
stencil_2d	1036	976	0.94	5.79

Table 5.4: LUT Usage for FTD and Legacy.

Kernel Name	Legacy LUTs	FTD LUTs	Legacy FFs	FTD FFs
get_tanh	10569	10452	3372	3221
jacobi_1d	17190	17224	3751	3751
kernel_2mm	20431	20078	5930	5134
kernel_3mm	26399	28168	6591	6741
matvec	578	492	588	433

Table 5.5: Results of synthesis for some benchmarks with no transparent buffers.

5.4 Conclusions

The results presented in this chapter demonstrate that *Fast Token Delivery*, combined with *Straight To The Queue*, leads to an average execution time improvement of 30%, a 19% reduction in flip-flops, and a 9% improvement in LUT usage. Given the heuristic buffering used for validation, these results should be considered successful.

There is, however, room for improvement. The following open issues should be addressed:

- **Fast Token Delivery Reliability**. As noted in Section 5.2, *Fast Token Delivery* passes all available tests in Dynamatic. However, it currently lacks a comprehensive set of unit tests that verify individual functionalities. The same applies to the analysis pass from Section 3.3. Implementing these tests is a crucial software engineering step for long-term maintainability of the compiler infrastructure.

- **Straight To The Queue Integration**. The interaction between the *Straight To The Queue* algorithm and the LSQ remains problematic. Currently, the algorithm fails in certain cases, which must be resolved. Among the proposed solutions in Section 5.2, one should be selected and integrated into the workflow, despite potential inefficiencies.

- **Buffering Algorithm**. Utilizing [11] or [9] is essential for a reliable comparison between *FTD* and *legacy*. Moreover, this approach would result in better circuits, reducing the need for transparent buffers and improving the critical path.

- **Out-of-Order Framework**. Once the above issues are addressed, Dynamatic's middle-end layer will be equipped to integrate the work from [14], a major advancement in dataflow circuit optimization.

Chapter 6

Thread-Level Parallelism in Dataflow Circuits

6.1 What is Thread-Level Parallelism?

Parallelization has always been a fundamental topic in computer architecture, since it can improve the available resource usage in a system.

It is common to distinguish three different kinds of parallelism [41]:

- *Data-Level Parallelism*: due to the availability of hardware resources of the same kind (vectorized processors or SIMD architectures) multiple data can be processed at once. As an example, instead of summing two vectors of 64 elements by iterating over each element, a vectorized adder can run all the additions in parallel.
- *Instruction-Level Parallelism*: multiple instructions are run in parallel and possibly disordered, thanks to an appropriate hazard-detection mechanism which maintains the original semantics of the program.
- *Thread-Level Parallelism*: multiple threads from a program execution can operate in parallel thanks to the availability of resources.

In this thesis, the main focus is on the third category (TLP). This has to do with MIMD (Multiple Instruction Multiple Data) systems, having multiple program counters and multiple units which allow for a parallelized execution.

There are different ways to exploit this methodology, as reported in [41]. *Parallel processing* is about «having multiple threads collaborating on a single task». On the contrary, *request-level parallelism* tries to execute multiple independent requests of the same kind at once.

TLP is exploited either by having many processors running in parallel, or by sharing the same processor for multiple threads in an interleaved fashion. This technique is called *multithreading*, and it will be the context of this research. In particular, the accelerator designed using Dynamatic, if inserted on an FPGA on a server, might request periodic activations, with multiple independent data to process. The possibility of handling multiple requests at the same time improves the resource utilization, and finally the throughput of the accelerator. This is almost equivalent to inserting the accelerator in a *streaming environment*, which refers to a system that is activated regularly at a given frequency. This is a standard assumption for a server, which gets a request to elaborate almost periodically.

Multithreading has different variations [41]. *Fine-grained multithreading* interleaves each of the available threads in each clock cycle. *Coarse-grained multithreading*, on the contrary, switches to a different thread only when a long stall might arise, to minimize the overhead due to the switching activity. However, *Simultaneous multithreading* (SMT) is the current most common variation, and it consists in having multiple threads executing in different stages of the pipeline of a dynamically scheduled CPU.

Since this work aims to introduce multiple tokens from different activations in the same dataflow circuit, the approach resembles SMT.

6.1.1 Issues with Thread-Level Parallelism

While SMT is usually a good way to speed up some programs, its advantage depends on the availability of resources and the structure of the program. Let's consider the code in Listing 6.1.

```
int div(int dividend, int divisor) {
    return dividend / divisor;
}
```

Listing 6.1: Example of TLP code.

This kernel implements a division between its two operands. Let's imagine that the hardware has an available divide unit in charge of the operations.

A long-latency unit is characterized by its *latency* L - the number of clock cycles required to complete a request - and its *initiation interval* II - the number of clock cycles to wait before another operand can be processed. Suppose the divisor mentioned above has an II of 1, which implies the possibility of running one operation per clock cycle. In that case, multiple instances of the function can run in parallel with the same underlying hardware.

However, if $II = L$ (thus only one division can be processed at one time) then no parallelism through the same unit is available under this condition.

Another common scenario in which the TLP cannot be used in such an immediate way is when, due to the execution speed, the results might be produced in a wrong order. Let's refer to Listing 6.2.

```
int sum(int n) {
    int i = 0;
    while(i < n)
        i++;
    return i;
}
```

Listing 6.2: Example of TLP code with reordering.

Let's suppose that, due to the available hardware, multiple instances of the thread/ kernel `sum` can be parallelized. The user first calls `sum(1000)`, then `sum(5)`. The first invocation of the function will require approximately 200 times the time of the second invocation. For this reason, the second request will likely finish before the first is completed. A proper runtime environment, made of a tagging and ordering mechanism, needs to be available to make sure that the correct result is delivered to the correct execution unit, without assuming the results remain in order.

6.2 TLP in Dataflow Circuits

6.2.1 Dataflow Circuits in a Streaming Environment

Dataflow circuits are good candidates for running multiple threads simultaneously, in SMT-fashion: due to their distributed nature, little should be done to allow multiple threads to flow at runtime. In particular, the idea is to develop a circuit on FPGA, and then insert such a circuit in a streaming environment. Referring again to Listing 6.1, the way it works seems clear if $II = 1$: each set of inputs are tokens that flow inside the divisor to produce a result. From an HLS perspective, it looks like inserting the body of the kernel around a for loop (Listing 6.3) with large N.

```
int streaming_div(int* dividend, int* divisor, int* result) {
    for(int i = 0; i < N; i++) {
        notify dividend[i] / divisor[i];
    }
}
```

Listing 6.3: Example of TLP code from an HLS perspective.

However, as the next section will demonstrate, dataflow circuits are not robust enough to handle multiple activations of the kernels at the same time without affecting correctness.

6.2.2 Performance Consequences

Let's consider the loop in Figure 6.1: there is a long latency operation, with an $II = 1$ and latency L. This is a FIFO with L slots. The number of iterations in the loop is N.

Let's say that both N and L are equal 2, and that, for each clock cycle, there is a token which might initiate the loop body (left-side of the μ component). By default, the μ is a `MUX`, allowing only one token to enter the loop body from outside, as long as there is an iteration still running. All the other incoming tokens are blocked on the μ left input (assuming, for instance, a large enough transparent buffer to store them).

A timeline of what happens in the loop is shown in Figure 6.2. Different loop bodies (instances of execution of the loop) are represented in different colors. The first row represents the token coming from the outside, accumulating in the FIFO; the second row shows the token regenerated at each iteration; the third row shows the tokens going out of the μ and entering the long-latency operation; the last row shows the tokens going out of the loop.

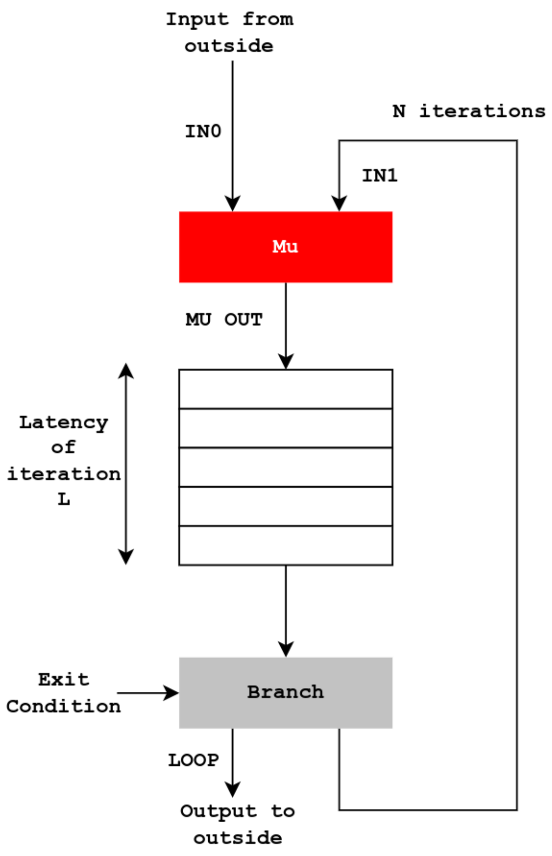


Figure 6.1: Example of loop in a dataflow circuit. The labels refer to the same used in Figure 6.2, Figure 6.3, Figure 6.4 and Figure 6.5.

As it was reasonable to guess, there is one token exiting the loop every 4 clock cycles ($N \cdot L = 2 \cdot 2 = 4$; in the real circuit, timing is also affected by the speed of the `MUX` condition, but this aspect will be covered in the next chapter more in depth). However, it is also clear that the long latency unit is under-utilized: out of the two available slots, only one of them is full at each clock cycle.

A possibility to solve the problem of maximizing the throughput and increasing the resource utilization is to implement the μ using a `MERGE`; this component accepts a token either from the left or from the right input, as long as there is capacity to accept more tokens in the loop body. The priority is given to the *right* side, so that previous instances of the loop body are terminated before new instances can start.

The corresponding timeline is shown in Figure 6.3. It becomes clear that, after a warm-up phase, not only the tokens are still provided to the outside with the correct order, but the throughput of the loop is also increased (2 tokens every 4 clock cycles).

It is thus evident that a conversion from a `MUX` to a `MERGE` can have some benefits for the throughput. However, this is not always the case. Let's remain in the scenario of a `MERGE` as loop header, but under the hypothesis of receiving a token from the left side of

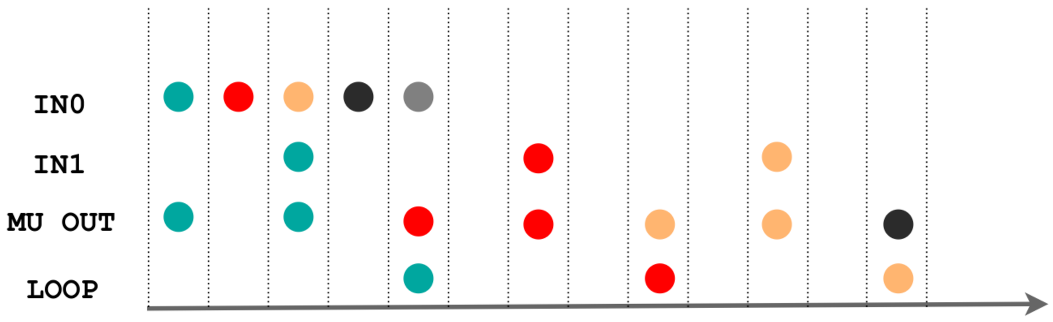


Figure 6.2: Timeline of loop body's tokens using a `MUX` as loop header.

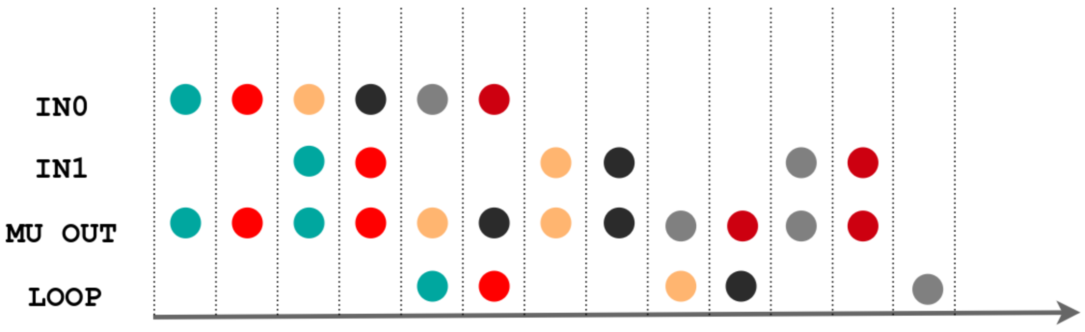


Figure 6.3: Timeline of loop body's tokens using a `MERGE` as loop header.

the μ every 6 clock cycles.

As it's evident from Figure 6.4, the `MERGE` is not able to improve the occupancy of the internal pipeline on its own, since there are not available tokens to use as soon as possible. In this case, the throughput of the loop is constrained by the speed at which it receives an activation token.

However, one can argue: why can't we use `MERGE`s only, since they provide benefits without having consequences? The second point of the question is not correct. Let's relax the hypothesis over the fixed number of iterations in the loop. There are two consecutive activations of the loop body, one which requires 4 iterations, one which requires 3 of them. The timeline is shown in Figure 6.5.

While the order of incoming tokens is *cyan* first and *red* second, the tokens exit the loop in the reverse order.

This is an out-of-order problem, which requires adequate infrastructure (provided by [14]) to guarantee correctness. The results from that work show an increase of area between $\sim 20\%$ and $\sim 250\%$.

For this reason, having a `MERGE` rather than a `MUX` affects the area requirement of the circuit, and should be carefully considered.

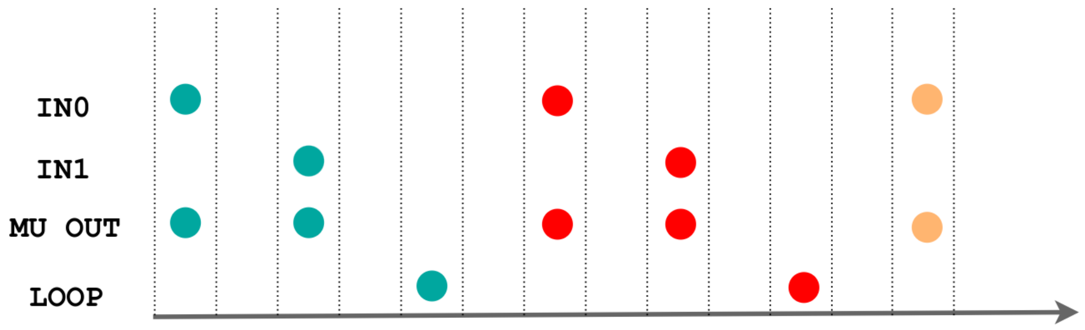


Figure 6.4: Timeline of loop body's tokens using a `MERGE` as loop header, but slower activations.

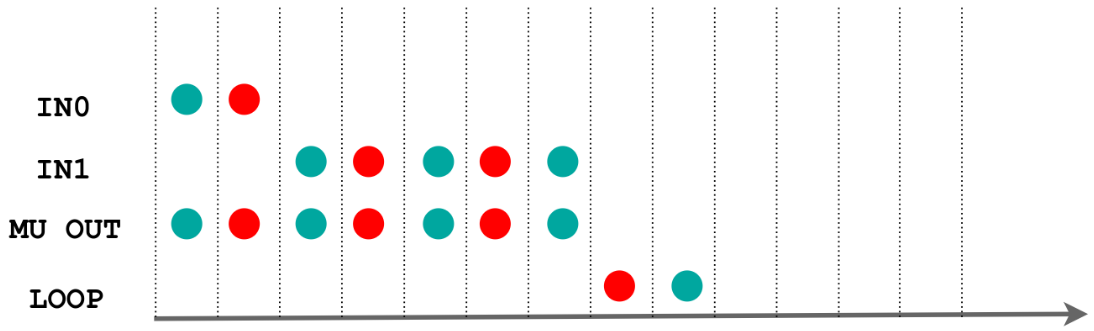


Figure 6.5: Timeline of loop body's tokens using a `MERGE` as loop header, but different number of iterations.

6.2.3 The Contribution of This Thesis

The decision of when to implement loop μs as `MERGE`s rather than `MUX`es should be taken only when there is a guarantee that this change will tangibly improve the throughput of the circuit in a given *streaming environment*, justifying any resource overhead associated with this. While [14] shows *how* to go out-of-order, it does not answer the problem of *where* to go out-of-order.

For this reason, the following chapter shows a methodology to adequately pick the set of `MUX`es that can be safely transformed into a `MERGE` to improve the performances, without adding resources if the throughput improvement is not guaranteed.

6.3 Previous Works

Cheng et. al. worked on loop-pipelining in HLS [18], which is exactly the open-ended question from the previous section: allowing multiple tokens to flow in the loop from Figure 6.1 without affecting the visible effects of the computation. Such a project has two objectives: on the one hand, lifting the C-slow pipelining technique, to introduce additional latency in the loop body (in the form of buffering) to improve the critical path while not altering the throughput; on the other hand, appropriately detect how many loop bodies can run in parallel without having memory hazards.

It demonstrates that almost a $3\times$ speed-up can be achieved with a minimal area overhead ($\sim 7\%$). The main focus is on the innermost loops of each loop-nest structure, adding an hardware header which limits the number of tokens which can flow in the body at once as long as it is safe to do so. To solve the ordering problem, a tag is associated with each token flowing, guaranteeing the correct order once the tokens are produced. This work, however, introduces the disordering mechanism to every innermost loop, without taking into account the possibility of it being not useful, as Subsection 6.2.2 shows.

A more general methodology for loop-pipelining and out-of-order execution in dataflow circuits is offered by [14]. It provides a framework to tag and reorder (or align in their terms) tokens in different areas of the circuit, together with an algorithm to make sure that functionalities are maintained. This framework allows better loop pipelining (or simultaneous multithreading) in loops by appropriately tagging and untagging tokens in exclusive local parts of the circuit, leaving the rest of the circuit untouched, keeping the outside execution in-order. As previously mentioned, however, this does not provide for a way to decide which areas of the circuit can benefit from it.

Chapter 7

Static Analysis for Thread-Level Parallelism in Dataflow Circuits

7.1 The *Occupancy* of a Channel

A circuit placed in a streaming environment, receiving periodically an activation signal, allows to consider the circuit as a network possibly reaching a steady state. This state corresponds to an equilibrium of the network, in which there is a regularity in the way tokens flow.

This is an ideal view of the network, possibly unreachable in the real-world. For instance, due to backpressure, a buffer can have a rate of input tokens that is higher than the rate at which tokens are released. This implies a moment in time in which the buffer will be full, propagating the backpressure uphill. This problem will be handled in Section 7.2.

At this stage, the focus is on a steady state, in which each channel in the circuit has a fixed number of tokens flowing per unit of time. Figure 7.1 shows, for some clock cycles, the value of the *valid* and *ready* signals of a channel. Due to the way the handshake protocol works, a token (the third row) flows only when both the *valid* and *ready* signals are active. As the figure shows, in this context there is one token flowing every three clock cycles. Thus, the average number of tokens on the channel per unit of time is $\frac{1}{3}$.

This concept, in dataflow circuits is referred to as *occupancy* or *throughput* [9], represented through the symbol Θ.

In the steady state, each channel in the circuit will have a fixed-value throughput. However, it should be noted that the throughput, being a single-value metric, cannot completely express the dynamic behavior of the circuit. For instance Figure 7.2 shows the throughput of a channel firing 3 tokens in three consecutive cycles and then waiting for 6: the throughput is $\frac{1}{3}$, same as in Figure 7.1. Notice that the throughput is upper-bounded by 1.

The objective of this chapter is to provide a model of the circuit which allows one to extract, for every channel in the circuit, the throughput at the steady state. This is coherent with the problem explained in the previous chapter: Having a way to estimate the final throughput of the circuit allows you to determine whether a `MERGE` used as loop

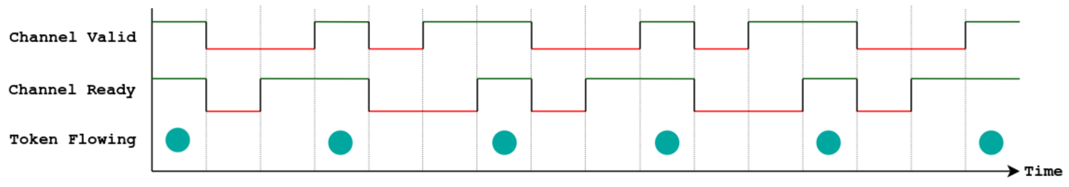


Figure 7.1: Graphical representation of the occupancy of a channel in the ideal case: there is one token flowing every 3 clock cycles, leading to an occupation of 0.33.

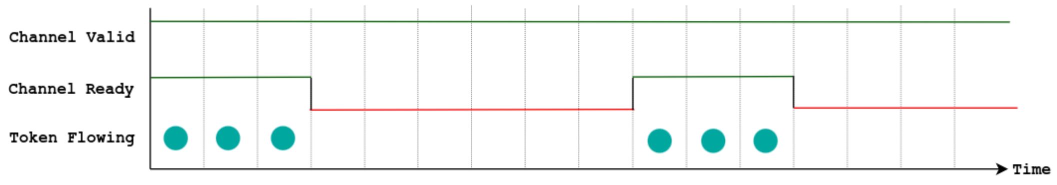


Figure 7.2: Graphical representation of the occupancy of a channel in the real case: three tokens flow every nine cycles, thus the occupancy is still 0.33.

header can improve the timing of the circuit, thus making the overhead of the out-of-order infrastructure worth it.

7.2 Assumptions and Context

This thesis focuses on providing a mathematical framework for running throughput analysis in dataflow circuits. In the future, a generalized framework will be developed, possibly using relevant models from the domain of *Network Calculus* [21]. The analytical approach that will be described has the advantage of showing that throughput analysis can be successfully obtained; as a disadvantage, it cannot analyze every *dataflow* structure.

To simplify the analysis of the circuit, and to specify the scope of the work, the following assumptions are taken into account:

- The kernel is inserted in a streaming environment, having a frequency of activation $\Theta_{external}$. This is equivalent to the throughput of the channels used as inputs of the circuit (arguments of the compiled function and *start* signal).
- Each channel has an infinite number of transparent buffers; as a consequence, back-pressure is (almost) never a factor. This is a very strong assumption over the system, and it is not feasible in a real circuit. Even a very large buffer might end up getting full in the long run, as explained in the previous section. However, the goal is to finally provide an ideal estimation of the throughput. Having infinite buffers corresponds to ideal buffering; then, a buffering algorithm should be in charge of

understanding where to put buffers and how to size them to reach the expected result, if possible.

- Memory dependencies are not considered. This means that the analysis gives for granted that no memory conflict can exist between multiple tokens flowing contemporaneously in the same loop. This problem was also taken into account in [14], by stating that such a memory conflict cannot be handled using the out-of-order framework for dataflow circuits. However, [18] aims at solving exactly this problem: at a later stage, this work can be adopted to guarantee correctness.
- Each loop has a fixed number of iterations N. This is a common assumption in static analysis related to CFGs, since it allows to know how many times a backedge is traversed. The value of N can be either known due to the loop structure (a for loop from 0 to $N-1$) or estimated by taking an average or a worst case value. In the circuits that are used to validate the system, a fixed number of iterations is always used. This guarantees that the order of tokens flowing out of the loop is always correct without the out-of-order infrastructure (refer to Figure 6.5). The main issue here is that such an infrastructure is not available in Dynamatic yet, thus it cannot be exploited for testing purposes. As a more general metric, it is possible to rely on the *probability of an edge to be traversed in the CFG*. For instance, at a loop exit, the probability of going back to the loop header will be $\frac{N-1}{N}$, while the probability of continuing is $\frac{1}{N}$.

7.2.1 The GSA Benefit

Fast Token Delivery allows to build a circuit which is way more optimized than the original one. In particular, being based on the GSA representation, the μ loop headers are well-distinct from the γ gates, and each of them already has the corresponding loop-exit condition as driver, rather than being connected to the *cmerge network*.

This allows, at the `handshake` level, to easily cluster loops and their loop headers, so that they can be handled separately without any inconvenience. Moreover, the out-of-order infrastructure is only coherent with the *Fast Token Delivery* methodology, thus TLP is not feasible without the work from Chapter 3 and Chapter 4.

7.3 Throughput Analysis Without Loops

In the assumption of having no loops in the kernel, the elastic circuit obtained from Dynamatic is an acyclic directed connected graph. In such a context, the output throughput of a component is dependent only on the throughput of the inputs, using an expression like $\mathbf{\Theta}_{out} = f(\mathbf{\Theta}_{in})$ (these are vectors, since there might be multiple inputs and multiple outputs, each of them with its throughput).

If such a formula is provided for each component, the lack of cycles guarantees a Breadth-First Search (BFS) to be a valid way to obtain all the throughputs, in a top-down fashion.

The following data structures are required:

- `T[channel]` stores the throughput for each channel in case it is already known. All the channels connected to the external inputs are initialized with $\Theta_{external}$.
- `R[component]` contains all the components whose input throughput is known. This means that the throughput of their output channels is ready to be computed. It is initialized with the components whose input throughput is in $T[channel]$.
- `L[component]` is the list of components that are left to be analyzed, due to the unavailability of some input data. It is initialized with all the components in G, except for those in R.

The BFS algorithm is detailed below.

1. $\Theta_{external}$ is given, together with the graph G representing the circuit: each node g_i is a component of a given type, while each edge c_{ij} is a channel from g_i to g_j. All the aforementioned data structures are properly initialized.
2. As long as R is not empty, pop an element g_i. The type of the component will provide a function f in the format $\mathbf{\Theta}_{out} = f(\mathbf{\Theta}_{in})$. Since g_i was in R, $\mathbf{\Theta}_{in}$ is known. Updated T with the values in $\mathbf{\Theta}_{out}$.
3. By having new throughput for new channels, some components might now become ready. Update R by moving components out of L.
4. At the end of the algorithm, L should be empty. T will provide the throughput of each channel.

As expressed in point 3 of the list above, each dataflow component is going to have an expression to compute the output throughput given the input throughput. Since the available dataflow components can all be seen as the list in Figure 2.7, providing a function f for each of them is enough to determine the throughput in the circuit. What follows is an analysis of these components. Notice that all these structures do not take into account the possibility of having backpressure from downhill, due to the assumption of infinite buffers.

- `JOIN`. This includes all the units which require the inputs to be synchronized to fire. This could be an arithmetic unit or a join itself. The components will be bottlenecked by the slowest of its N inputs, thus:

$$\Theta_{join} = min(\Theta_1, \Theta_2, \ldots, \Theta_N)$$

- `BUFFER`. Whichever buffer is considered, both opaque and transparent, will propagate the token at the same frequency of the input Θ_{in} (at most, with a one-cycle latency):

$$\Theta_{buffer} = \Theta_{in}$$

- `LLC`. This is a long-latency operation with an $II \geq 1$. In this case, the backpressure cannot be avoided, since it is due to unavailability of resources. Considering N inputs (with possibly $N = 1$):

  $$\Theta_{LLC} = \frac{min(\Theta_1, \Theta_2, \ldots, \Theta_N)}{II}$$

  For instance, consider a 10-cycle divisor unpipelined which receives activation tokens at every clock cycle. Due to its nature, it will not be able to produce a token faster than 10 cycles, thus the output throughput will be $\frac{1}{10} = 0.1$.

- `MUX`. A `MUX` merges two channels (true and false input) while synchronizing with the condition token. This means that a token cannot move on if the condition is slow. This is explicated by the following expression:

  $$\Theta_{mux} = f_T \cdot min(\theta_C, \theta_T) + f_F \cdot min(\theta_C, \theta_F)$$

  Not only are the inputs synchronized, but they are weighted according to the probability of picking either the true or false side. For instance, if the left side is very fast, but it is seldom picked, then it will not impact much on the output throughput. Being the probabilities of picking the two sides, $f_T + f_F = 1$. This condition guarantees $\Theta_{mux} \leq 1$, as the throughput should be. The probabilities involved are related to the way the CFG is traversed, so they are known according to the assumptions.

- `MERGE`. Since the `MERGE` can arbitrarily pick on the two sides, the output is just the sum of the N input throughputs. However, in this case the sum could be higher than 1 (if all the inputs are very fast), thus the upper-bound needs to be explicated.

  $$\Theta_{merge} = min(1, \sum_i \Theta_i)$$

  The same expression can be used for a `CMERGE`, considering also the *index* output.

- `FORK`. Each of the N outputs of a fork has the same throughput of the input, Θ_{in}:

  $$\Theta^i_{fork} = \Theta_{in} \quad \forall \text{ output } i$$

- `BRANCH`. The branch mirrors the `MUX` case. On one hand, it needs to synchronize the input Θ_{in} with the condition token Θ_c; on the other hand, the outputs have different probabilities to be picked:

  $$\Theta_T = f_T \cdot min(\Theta_c, \Theta_{in})$$
  $$\Theta_F = f_F \cdot min(\Theta_c, \Theta_{in})$$

While these expressions have been obtained analytically, they are meant to approximate the model of each dataflow component when no backpressure is involved. However, it is reasonable to think that more formal methods can be found to obtain the same results. While no further analysis has been done, being too advanced for the scope of this thesis, [21] can be a reasonable tool.

7.3.1 An Example of Throughput Analysis Without Loops

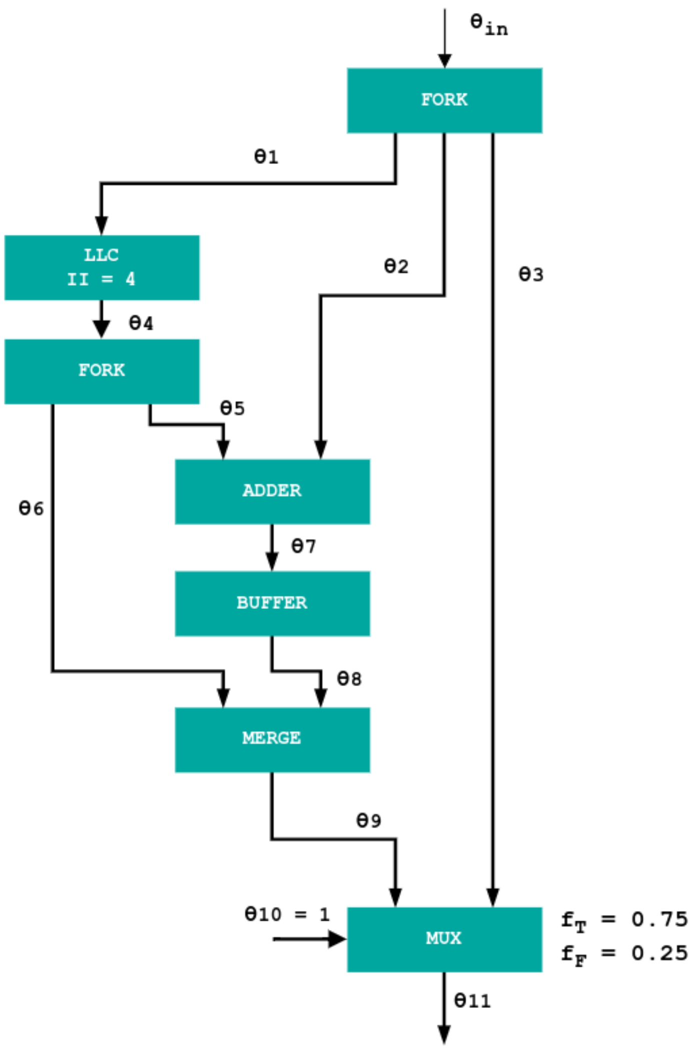


Figure 7.3: Example for a throughput analysis without loops

Figure 7.3 shows a dataflow circuit without any loop in the dataflow graph. Out of this circuit, the algorithm can be run by considering $\Theta_{in} = 1$ and the expressions for each component.

$$
\begin{aligned}
\Theta_1 &= \Theta_{in} = 1 \\
\Theta_2 &= \Theta_{in} = 1 \\
\Theta_3 &= \Theta_{in} = 1 \\
\Theta_4 &= \Theta_1/4 = 0.25 \\
\Theta_5 &= \Theta_4 = 0.25 \\
\Theta_6 &= \Theta_4 = 0.25 \\
\Theta_7 &= min(\Theta_5, \Theta_2) = min(0.25, 1) = 0.25 \\
\Theta_8 &= \Theta_7 = 0.25 \\
\Theta_9 &= min(1, \Theta_6 + \Theta_8) = min(1, 0.5) = 0.5 \\
\Theta_{10} &= 1 \\
\Theta_{11} &= f_T \cdot min(\theta_{10}, \theta_9) + f_F \cdot min(\theta_{10}, \theta_3) = 0.75 \cdot 0.5 + 0.25 \cdot 1 = 0.625
\end{aligned}
$$

7.4 Throughput Analysis With Loops

The above approach can be utilized with sub-circuits having no backedges or loops. However, when a loop is present, the same mechanism cannot be utilized, since the behavior of a single component is not enough to determine the behavior of the loop.

Let's refer to the simplified version of a loop as proposed in the previous chapter, also shown in Figure 7.4. The current goal is to determine Θ_{out}. This value will depend on the input throughput of the loop, Θ_{in}, but also on the latency of each iteration and the number of iterations. To understand why this is the case, consider the `MUX` scenario: once the activation token starts a loop batch, a new token will flow on the output of the μ after L clock cycles, and then again for N times until another loop batch can be activated. For this reason, the calculations presented in Section 7.3 are not enough and a more general analytical analysis should be done. As Subsection 7.6.2 will show, such analysis is also validated experimentally.

7.4.1 `MUX`es as Loop Headers

In the `MUX` scenario, one token can flow in the pipeline at one time. When a token goes out of μ, then you have to wait at least L clock cycles to get a new token, being L the latency of the pipeline. As a consequence, $\frac{1}{L}$ is a lower bound of the throughput of such channel. However, this is the case if and only if the μ is always running: on the contrary, if not enough tokens are available coming from outside the loop, the throughput will be lower. Generally speaking, for each token coming from the left side, N tokens will flow out of μ. For this reason, a first approximation of Θ_{out} for a loop header `MUX` is

$$\Theta_{out}^{mux} = min(N \cdot \Theta_{in}, \frac{1}{L})$$

However, this expression does not take into account the speed at which the exit loop condition (which drives the `MUX`) is produced. If the condition takes little time to be

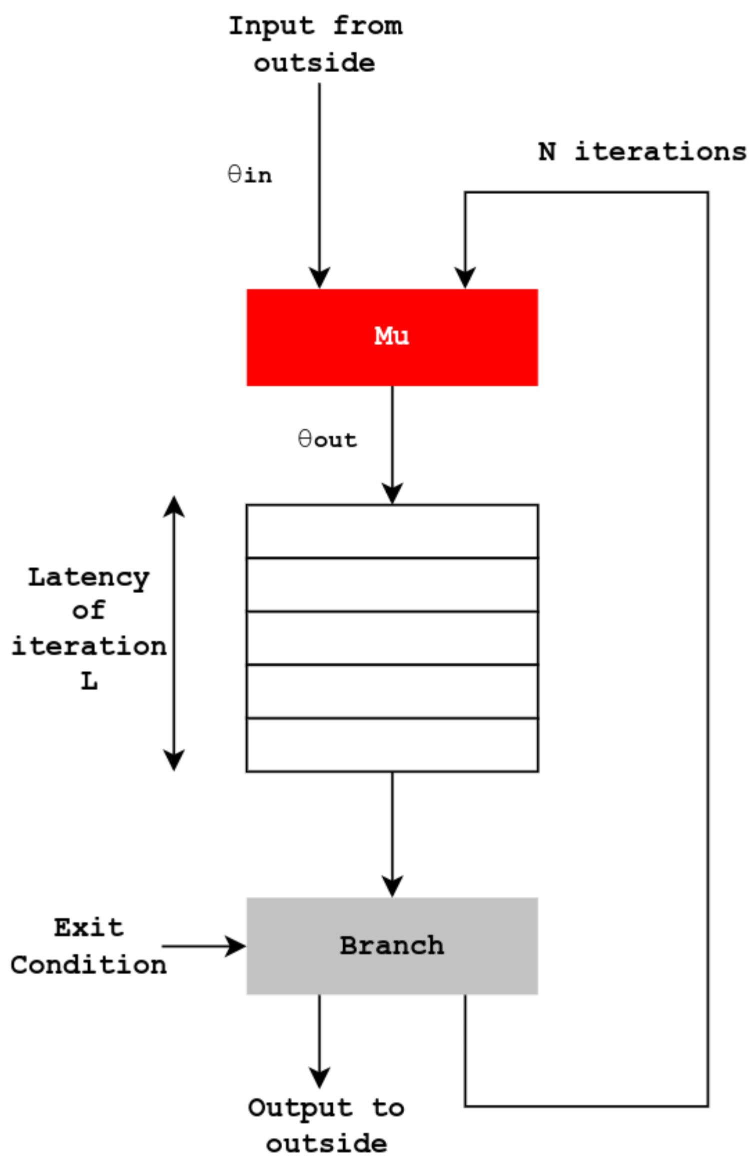


Figure 7.4: Simplified view of a loop.

computed (ideally, one clock cycle) then *the first iteration of the previous loop can start one clock cycle after the last iteration of the previous loop*. The result is also highlighted by [18]. This implies having two tokens flowing in the loop body at the same clock cycle, leading to a higher throughput. Since the condition is usually immediate to compute (made of a combinatorial path of an adder and a comparator), in almost all the realistic circuits an improved version of the formula is

$$\Theta_{out}^{mux} = min(N \cdot \Theta_{in}, \frac{N}{L \cdot (N-1)})$$

While the first part of the expression takes into account a *slow loop activation*, the second part tells that N tokens flow in the circuit every $L \cdot (N-1)$ clock cycles. Notice that, if $N \gg 1$, then the second part still depends only on L as a term, since $\frac{N}{N-1} \sim 1$.

Since, at most, one token can be present at one time, having pipelined or unpipelined units (thus units with $II > 1$) does not impact the resulting throughput.

7.4.2 MERGE as Loop Headers with Operations Having II = 1

When a MERGE is present as loop header, many tokens can be accepted from the outside, as long as they are available and there is space within the circuit. Let's consider the case of the loop body as a long pipeline (possibly made by many components in cascade) with an II of 1 each.

Once all the s slots of the cascade of components are filled, then no more tokens can be accepted: the s tokens already inside will iterate N times in the circuit until their loop batches are completed. At that point, if available, s new tokens can enter.

To express Θ_{out} in this case, it should be noted that if the pipeline is full, then a token will flow through the μ per clock cycle (either from the left or right side), and its throughput will be one. However, if the left side is not fast enough to provide tokens to fill the pipeline, the throughput will be bottlenecked by this limitation. As a consequence:

$$\Theta_{out}^{merge} = min(N \cdot \Theta_{in}, 1)$$

This is evident in Figure 6.3 from previous chapter: there is always a token flowing through the μ if enough tokens are provided; on the contrary, Figure 6.4 depicts an μ output throughput of $\frac{1}{3}$, as expected from the last expression.

7.4.3 MERGE as Loop Header with Operations Having II > 1

So far, there was no numerical distinction between the number of tokens which could flow in the pipeline (s) and the latency of the pipeline (L), having no backpressure and $II = 1$ for all the components.

Having a component with an $II > 1$ and a latency L, it can accommodate at most $\frac{L}{II}$ tokens at the same time. For this reason, numerically speaking, $s \leq L$.

While having infinite transparent buffers on each channel guarantees no backpressure on *fork-join* paths, an operation with $II > 1$ will always generate backpressure, since II clock cycles are to be waited before a token can proceed. Such backpressure is then propagated upstream.

Consider Figure 7.5. Let's say that the loop body is full (each component has a token inside) and that the last component just terminated. The token from the second component will flow down, and so will the token in the first component. Also, let's say there is a new token entering the first component (either from the outside or from the backedge). After one clock cycle, the first component is done computing. However, due to the second component having $II = 2$, it needs to wait with a *valid* signal on. After another

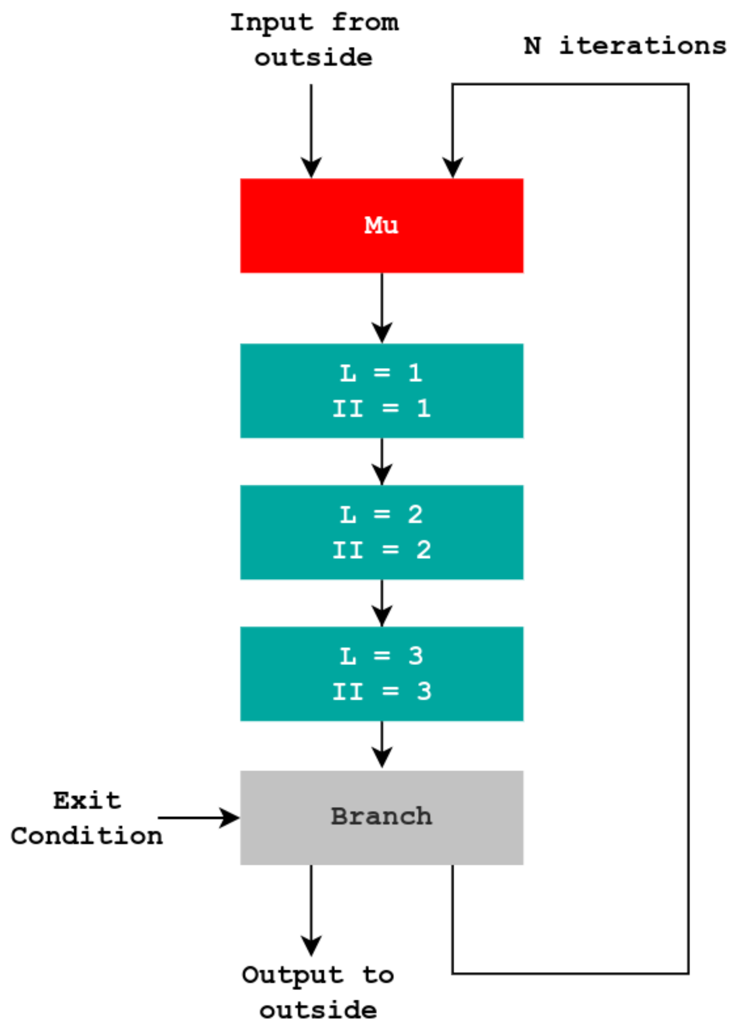


Figure 7.5: Operations with $II > 1$ forcing backpressure in the loop body.

clock cycle, the second component gets *valid* as well; though, the bottom component needs one clock cycle more. This way, tokens can move only every 3 clock cycles, bottlenecked by the speed of the component with larger II. A graphical representation of the timeline is shown in Figure 7.6

When the pipeline is full, a token entering the pipeline requires, for each iteration, the time for the components themselves (L) and also the bottleneck time, expressed by the difference between the maximum II of the loop body and the II of the other components. This time is then $L + \sum_i (II_{max} - II_i)$. Notice that, according to this analytical reasoning, if all the components have the same II, then no backpressure needs to be taken into account, which is coherent with what you would expect. When the pipeline is full, s tokens will flow through μ by the time one iteration is done. For this

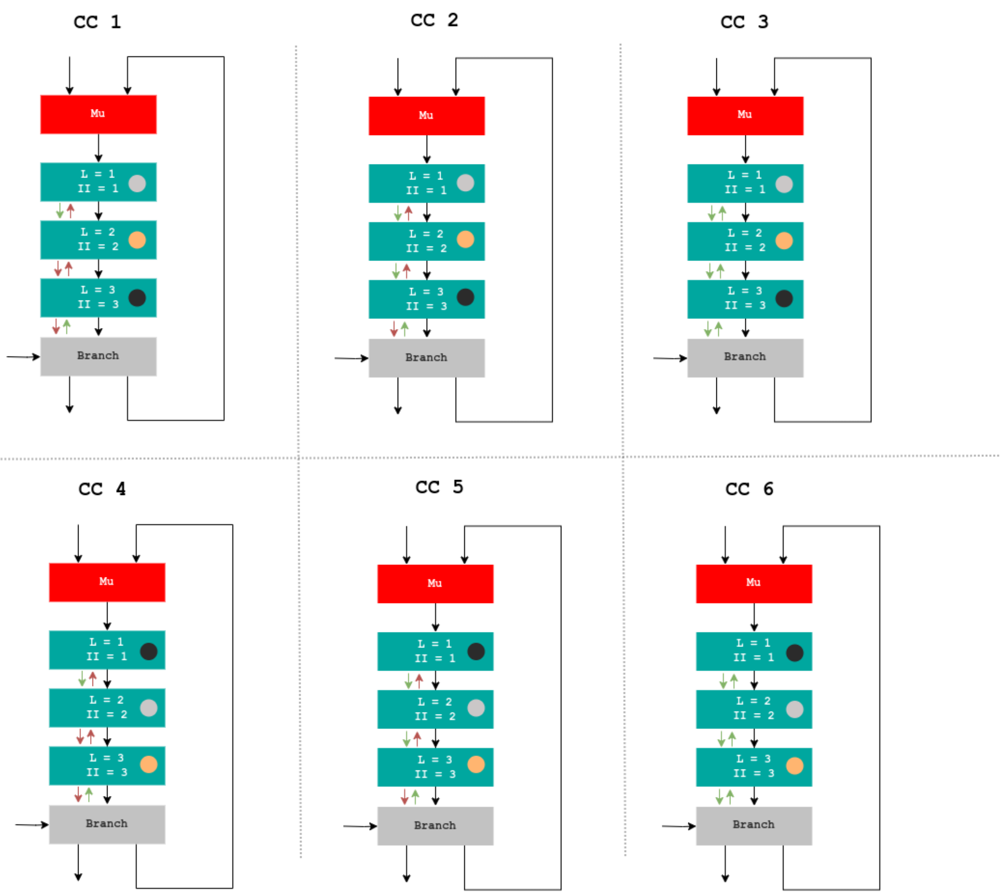


Figure 7.6: Example of bottleneck in loop bodies with components having $II > 1$. 6 clock cycles are represented; for each component in the loop, their respective *valid* (down arrow) and *ready* (up arrow) signals are represented. Different tokens currently handled are represented through different colors within the components.

reason, the expression of the `MERGE` becomes:

$$\Theta_{out}^{merge} = min(N \cdot \Theta_{in}, \frac{s}{L + \sum_i (II_{max} - II_i)})$$

If $II = 1$ for each component, then $s = L$ and the left part of the expression is 1 again, coherently with the previous result $min(N \cdot \Theta_{in}, 1)$

7.4.4 Multiple MU in One Loop

In general, the simplified view of the loop that has been proposed does not suit the reality of a loop as it can appear in a circuit, due to the multiple μs in its loop header and the numerous cycles within it.

Figure 7.7 shows an example of a loop which divides a number by a constant for N times. Two μ have the purpose of regenerating a token as long as it is necessary; the

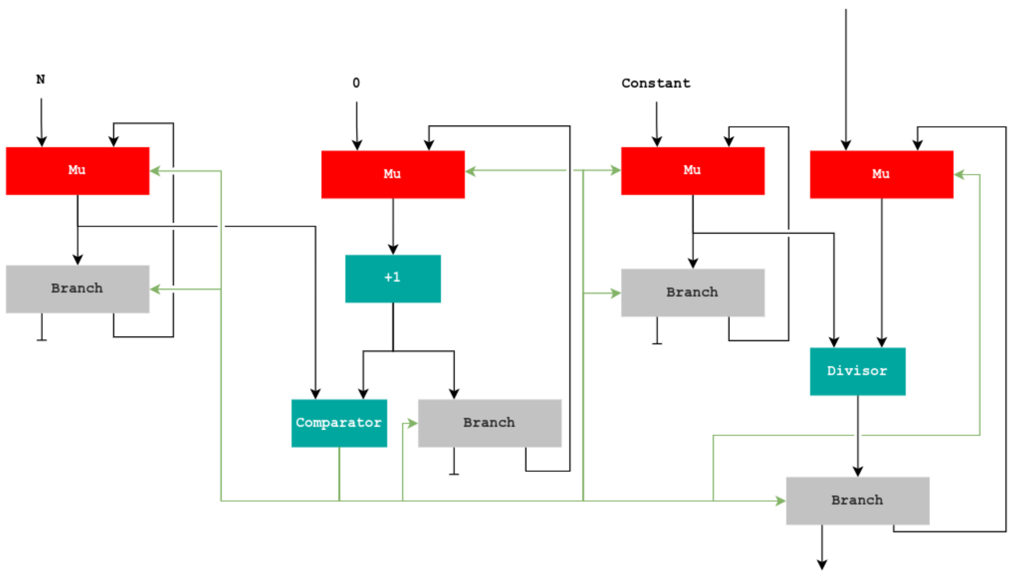


Figure 7.7: Example of multiple μs in a loop. For the sake of simplicity, `FORK`s are omitted.

other 2 are in charge of modifying the value of the divided token (right-most μ) and the iteration variable (second μ from left).

In general, a definitive solution to solve the problem of the complex interaction between components was not found, especially in the context of $II > 1$, due to the forced backpressure this produces. However, a correct approximation can be found if $II = 1$ for all the components.

As Subsection 7.6.1 will show, all the μ of one loop header are going to be connected, driven by the one with a slower iteration timing. For this reason, it is necessary to go through every possible cycle starting from a μ of one loop header and ending into another μ (possibly the starting one) of the same loop header. Out of these cycles, the one with the highest latency is considered. If multiple cycles share the same value of L, the one with less slots is taken, since $\Theta_{out} \propto s$. Using this path to compute the formula of the `MERGE` case will provide the throughput of all the μ for the same loop header.

7.4.5 Nested Loops

One might ask why so far a lot of effort was put into considering components with $II > 1$. It is interesting to note that Dynamatic has no such component among the available ones. However, nested loops can be modeled as long latency components when computing the throughput for μs: they require a fixed number of clock cycles to elaborate a token (the latency L) and have a fixed number of tokens that can process at one time (s). In particular, if the loop has `MUX`es as loop header, then there can be at most one token ($s = 1$), otherwise there can be as many as the number of slots available.

The latency of a loop is given by the latency of the longest cycle in the loop itself, as expressed in the previous section, multiplied by the number of iterations N minus one. This is a consequence of being able to start a new loop batch at the beginning of the last iteration from the previous batch, as stated before. The II of the loop is thus:

$$II_{loop} = \frac{(N-1) \cdot L_{\textbf{longest cycle}}}{s}$$

Taking into account such considerations, a loop can be seen as a black box with a given latency and initiation interval. This is useful for applying the expressions from the previous sections, to find the output throughput of a μ that feeds an inner loop.

7.5 Combining All Together

The previous section provided a way to study loops and μ gates in the handshake circuit, to extract the throughput of most of the channels. Now, all the steps should be put together to define an algorithm which is in charge of determining the throughput for each channel given the input throughput and the circuit.

In particular, such an algorithm can solve the problem of determining when going out-of-order is beneficial: by trying all the possible pairs of combinations between `MUX` and `MERGE` as loop headers, the configuration having a better output throughput can be obtained. This approach covers all the possible points in the design space, which are exponential with respect to the number of loops N.

The following data structures are adopted:

- `is_merge[loop]`. For each loop, stores whether the loop has merges as loop headers or not. This is an input of the algorithm.
- `N[loop]`. Stores, for each loop, the number of iterations it runs. This is an input of the algorithm.
- $\Theta_{external}$ The activation frequency of the kernel. This is an input of the algorithm.
- `loop_info[loop]`. For each loop, stores the information related to the longest cycle found by the algorithm. In particular, it stores the latency L of the path, the number of available slots s and the II_i of all the elements in the path. If the path has a nested loop inside, then the loop is considered as a black box with its latency, number of elements, and II.
- `Throughput[channel]`. This stores, for each channel, the computed throughput. It is initialized with all the channels connected to the inputs of the kernel with the value $\Theta_{external}$.
- `Ready[component]`. Queue of operations that are ready to be analyzed, since the throughput of all the inputs is known.

The algorithm starts by filling, for each loop, the information related to the structure `loop_info`. Since, with nested structures, the information depends on the characteristics of the nested loops, the loops need to be covered in post-order traversal, prioritizing innermost loops. This way, all the information related to the loops is extracted. Then, it uses the same top-down methodology described in Section 7.3, by also incorporating the μ expressions. The detailed algorithm can be found in Algorithm 1.

Algorithm 1 Complete algorithm for throughput analysis.

```
1:  for each loop $l$ in the circuit, in a post-order traversal do
2:    for each cycle $c$ starting from a $\mu$ of $l$ and ending to a $\mu$ of $l$ do
3:      Count the total latency $L_{l,c}$ by summing single latencies;
4:      Count the total number of slots $s_{l,c}$ by summing the slots;
5:      Count the initiation interval of each component in the cycle, $II_{l,c,i}$
6:      if the path enters a nested loop $\hat{l}$ then
7:        The latency is $\texttt{is\_loop}[\hat{l}][latency] \cdot \texttt{N}[\hat{l}]$;
8:        The slots are $\texttt{is\_merge}[\hat{l}]?\texttt{loop\_info}[\hat{l}][slot] : 1$;
9:        The initiation interval is $\frac{\texttt{loop\_info}[\hat{l}][latency]\cdot(\texttt{N}[\hat{l}]-1)}{\texttt{is\_merge}[\hat{l}]?\texttt{loop\_info}[\hat{l}][slot]:1}$;
10:     end if
11:   end for
12:   Pick the path with the longest latency and smallest number of slots;
13:   Update loop_info[$l$] accordingly.
14: end for
15: while $Ready$ is not empty do
16:   Pop an operation $op$ from $Ready$.
17:   Compute its output channel throughput using the methodologies expressed in the
      previous sections and add it to Throughput.
18:   for each operation connected to its output do
19:     if all their input channels are in $Throughput$ then
20:       Add the operation to $Ready$.
21:     end if
22:   end for
23: end while
24: if there are values without computed throughput then
25:   return Error
26: else
27:   return Success(Throughput)
28: end if
```

7.6 Validation

It is important to stress again that the aim of this work is not to provide a reliable and complete static throughput analysis, but to show that the task can be achieved. The following experiments will show the validity of such methodology. Further research will later expand the analysis.

7.6.1 Circuit Modifications

As previously mentioned, since the *out-of-order* infrastructure is not available in MLIR Dynamatic yet, the testing mechanism had to be done without it. To guarantee correctness, having a fixed number of iterations is enough (the situation from Figure 6.5 cannot be tested). Then, all the problems related to memory interconnections have been solved by not having memories in the tested circuits, according to the initial assumption.

The *streaming environment* is also difficult to recreate, due to the current limitations of the circuit interface of Dynamatic when it comes to integrating the generated hardware into another system. For this reason, such an environment has been made by introducing a for-loop outside of the kernel itself, sending regularly some tokens to activate the computation.

A large number of transparent buffers (10000) are inserted at the end of each fork output and after each opaque buffer, to fulfill the assumption of having no backpressure. In all the provided test cases, the number is high enough to be in the *infinite buffer* scenario.

Another point consists in the conversion from `MUX` to `MERGE`. Since any loop has multiple μs in its loop header, an appropriate transformation should be done to guarantee correctness and avoid deadlock. If all the `MUX`es were to be translated into a `MERGE`, then one of them might pick a token from the right side in the same clock cycle in which another `MERGE` picks a token from the left side. This ends up with some tokens that were not supposed to be combined, leading to a wrong execution result.

This happens because, in general, different μs have different speeds in the single iteration. Refer again to Figure 7.7: the first three μ from the left can handle an iteration per clock cycle, while the right-most requires multiple clock cycles. When the *out-of-order* infrastructure is missing, all the μ have to behave like the slowest one, so like the one having a longer cycle to compute the token for the next iteration. This is the reason why Subsection 7.4.4 states that the throughput of each μ is identical to the one of the slowest μ.

From a dataflow circuit perspective, this can be done by picking the *slowest `MUX`* (the one which is the ending point of the cycle stored in `loop_info`) and transform it into a `CMERGE`. The index output of this component will drive the control of all the other `MUX`es. An example of such transformation, applied to Figure 7.7, is shown in Figure 7.8 with the blue arrows.

Once the circuit is simulated and the functionality is guaranteed, the metrics are extracted from the waveforms, by checking how many clock cycles a channel has both the *valid* and *ready* signal high together. Dynamatic already had a way to parse the waveforms out of a `wlf` file; however, the exact semantics of the handshake channel (a token flowing under those conditions) needed to be implemented.

7.6.2 Results

Experiment 1

The first kernel used to experimentally validate the technique is shown in Listing 7.1. It is made of two consecutive loops: `L2`, having an internal pipeline with a latency of 3, and

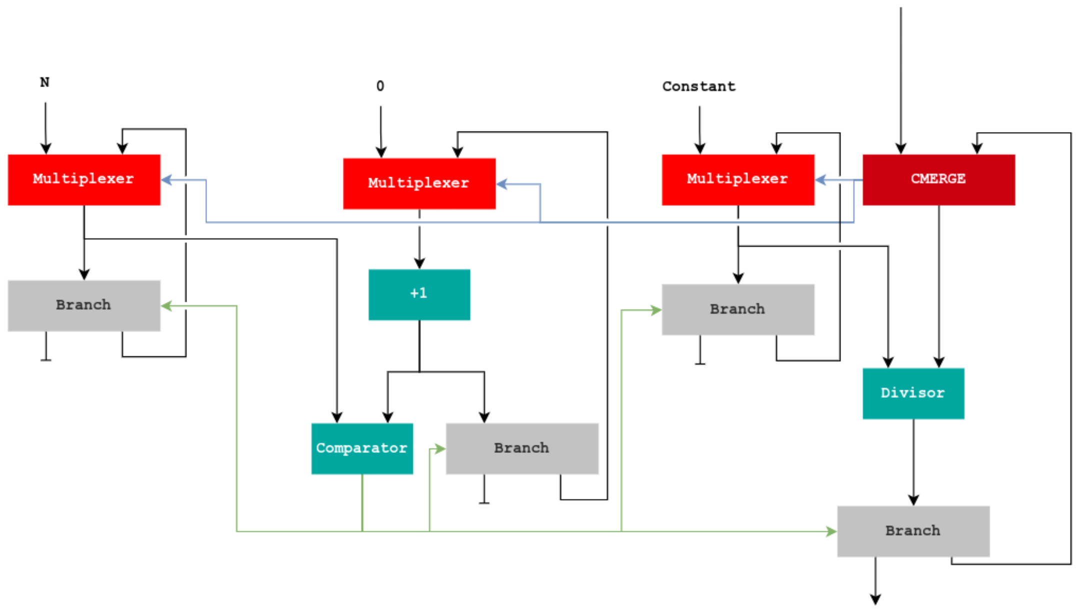


Figure 7.8: Transformation applied to a circuit to guarantee correctness while having a `MERGE` as loop header.

`L3`, having a latency of 5. The first loop requires 12 iterations to run, while the second loop requires 15. The outer loop, `L1`, just has the purpose of inserting the body into a streaming environment (notice that there are no loop carried dependencies within it. It injects in the kernel 200 tokens with $II = 1$ ($\Theta_{external} = 1$).

Also, all the errors are computed as $(\text{Expected} - \text{Measure})/\text{Expected}$, to quantify how much the model is distant from the real value.

```c
#define M 200
#define N 12
#define P 15

void experiment_1() {

  int sum;

  L1: for (unsigned i = 0; i < M; i++) {
    sum = 0;

    // Delay iteration (addi1) : 3
    // Number of iterations: 12
    L2: for (unsigned j = 0; j < N; j++)
        sum += (i + j);

    // Delay iteration (addi4) : 5
    // Number of iterations: 15
```

```
        L3: for (unsigned k = 0; k < P; k++)
            sum += (i + k);

    }
}
```

Listing 7.1: Experiment 1 for throughput analysis.

The kernel will be considered in 4 scenarios: no conversion, conversion of `L2` to a `MERGE` only, conversion of `L3` to a `MERGE` only, conversion of both loops. The metrics that are used for validation are: the throughput of the loop-header of `L2`, called Θ^{L2}_{out}; the throughput of the loop-header of `L3`, called Θ^{L3}_{out}; the throughput of the production of tokens from `L2` (true output of the final `BRANCH`), called Θ^{L2}_{loop}; the throughput of the production of tokens from `L3`, called Θ^{L3}_{loop}. With respect to Figure 7.8, Θ_{out} is the throughput of the output channel of each μ, while Θ_{loop} is the throughput of the output channel of the bottom-right branch.

It is worth noting that the result produced by `L3` can also be considered as the result from the kernel. For this reason, increasing Θ^{L3}_{loop} implies having a kernel which can provide results faster to the user.

The results are shown in Table 7.1. These are obtained by running the algorithm explained in the previous sections. Let's consider the first version, with no conversions. The input throughput of `L2` is 1, being connected directly to the streaming environment. This is Θ_{in}. Then, $\Theta^{L2}_{out} = min(N \cdot \Theta_{external}, \frac{N}{L \cdot (N-1)}) = min(12, \frac{12}{11 \cdot 3}) \approx 0.363$. This value can be used within the loop to compute the throughput of the components in the pipeline. When considering the output of the final branch, its throughput will be $\Theta^{L2}_{out} = \frac{0.363}{12} \approx 0.030$, which is also the throughput at which `L3` receives activation tokens. Thus $\Theta^{L3}_{out} = min(15 \cdot 0.030, \frac{15}{5 \cdot 14}) \approx min(0.45, 0.214) = 0.214$. For the same reason as before, $\Theta^{L3}_{out} = \frac{0.214}{15} \approx 0.14$

The results are promising, due to the very small error between the measurements and the expectations. On the contrary, the better resource usage shows that an improvement in Θ^{L3}_{out} is reachable, in this case by almost a factor of 4.7×.

Experiment 2

The previous example corresponded to a simple scenario with no loop nested and no operations with II larger than 1. On the contrary, Listing 7.2 shows a more complex kernel, having a nested loop.

```
#define M 200
#define N 12
#define P 15

void experiment_2() {
    int sum;
    L1: for (unsigned i = 0; i < M; i++) {
        sum = 0;

        L2: for (unsigned j = 0; j < N; j++) {
```

Table 7.1: Results of the throughput analysis from the first experiment.

	Θ^{L2}_{out}	Θ^{L3}_{out}	Θ^{L2}_{loop}	Θ^{L3}_{loop}
1. No Conversion				
Measured	0.353	0.211	0.029	0.014
Expected	0.363	0.214	0.030	0.014
Error (%)	2.75	1.40	3.33	0
2. Conversion on `L2`				
Measured	1	0.211	0.084	0.014
Expected	1	0.214	0.083	0.014
Error (%)	0	1.40	1.20	0
3. Conversion on `L3`				
Measured	0.353	0.439	0.029	0.029
Expected	0.363	0.454	0.030	0.030
Error (%)	2.75	3.30	3.33	3.33
4. Conversion on `L2` and `L3`				
Measured	1	1	0.084	0.067
Expected	1	1	0.083	0.067
Error (%)	0	0	1.2	0

```

            // Latency of this addition: 5
            sum += 2;

            L3: for (unsigned k = 0; k < P; k++) {

                // Latency of this addition: 10
                sum += (i + k);
            }
        }
    }
}
```

Listing 7.2: Experiment 2 for throughput analysis.

`L1` works again as a streaming environment; `L2` is the outer loop, so the value produced by it (`sum`) will be provided outside, representing the throughput of the kernel.

The results are shown in Table 7.2. Again, all the possible pairs of conversions between `L2` and `L3` are considered. It is clear that, compared to the previous experiment, the errors are larger, although always smaller than 10%. In particular, the average error across all the cases is $\sim 5\%$ (without considering the exact pairs, otherwise it would be ~ 3.5).

Most of the time, the estimation of the throughput is larger than the correct one, meaning that some factors are not taken into account when analyzing the circuit.

Just as an example, let's see how the estimation of the throughput has been done when `L2` is converted. Since `L3` is still with a `MUX`, it has a latency of $11 \cdot 15 = 165$, it has only one slot and its II is $11 \cdot 14 = 154$. The outer loop receives again a token every clock cycle, so $\Theta_{external} = 1$. Since it is a `MERGE`, the corresponding formula should be used to find the Θ_{out}^{L2}: $\Theta_{out}^{merge} = min(N \cdot \Theta_{in}, \frac{s}{L+\sum_i (II_{max} - II_i)})$. The number of slots is $6 + 1 = 7$, because of the 5-slots pipeline of the adder, the opaque buffer after the `MERGE` and the loop of `L3`. The latency is $L = 6 + 165 = 171$, while $II_{max} = 154$. All the 6 elements of the pipeline have an II of 1. For this reason, $\Theta_{out}^{merge} = min(12 \cdot 1, \frac{7}{171+6\cdot(154-1)}) \approx 0.00642$. For `L3`, the `MUX` scenario is used, having Θ_{out}^{L2} as input throughput: $\Theta_{out}^{L2} = min(15 \cdot 0.00642, \frac{15}{11\cdot 14}) \approx min(0.0963, 0.0974) = 0.0963$. $\Theta_{loop}^{L3} = \Theta_{out}^{L2}/15 \approx 0.00642$. Finally, the output throughput of the external loop can be found by considering that the output branch from $L2$ is connected to the output branch of $L3$, thus $\Theta_{loop}^{L2} = \Theta_{loop}^{L3}/12 = 0.00642/12 \approx 0.00053$.

Table 7.2: Results of the throughput analysis from the second experiment.

	Θ_{out}^{L2}	Θ_{out}^{L3}	Θ_{loop}^{L2}	Θ_{loop}^{L3}
1. No Conversion				
Measured	0.005 90	0.088 49	0.000 49	0.005 90
Expected	0.006 26	0.097 46	0.000 52	0.006 49
Error (%)	5.70	9.20	5.77	8.67
2. Conversion on `L2`				
Measured	0.007 03	0.096 80	0.000 54	0.006 49
Expected	0.006 42	0.096 30	0.000 53	0.006 42
Error (%)	9.50	0.5	1.89	1.09
3. Conversion on `L3`				
Measured	0.006 38	0.095 60	0.000 53	0.006 38
Expected	0.006 38	0.095 69	0.000 53	0.006 38
Error (%)	0	0.09	0	0
4. Conversion on `L2` and `L3`				
Measured	0.072 46	1	0.005 53	0.066 67
Expected	0.066 67	1	0.005 55	0.066 67
Error (%)	8.68	0	0	0

7.7 What is Missing and Future Work

The current result shows a methodology which can be adopted to start the throughput analysis of dataflow circuits. While promising, it still lacks the capability of fully managing all the possibilities arising when obtaining dataflow circuits through the *Fast Token Delivery* methodology.

In general, the next step should be to integrate such results with the current literature in throughput analysis in networks (since the circuits in Dynamatic can be seen as networks having packets flowing through *routers* - the single components - each of them having an appropriate semantics). It is reasonable to think that [21] might be a starting point. It should be noted that, in this context, the inner dynamic behavior of the circuit will probably never allow for a full model to exist. The solution needs to set a maximum error percentage which is considered acceptable.

Afterward, the assumption about the memory dependencies should be relaxed, with two possible alternatives: imposing in-order computation whenever a memory dependency exists; adopting [18] to limit the tokens flowing in a loop body (although this would introduce new modifications in the mathematical framework of the throughput analysis).

Once the model is verified, the *out-of-order* system from [14] should be integrated, by correctly identifying which points in the circuit require it. For instance, loops with a fixed number of iterations do not need it, as the examples from this chapter show. However, a trade-off between area increase and throughput improvement should be taken into account: how much area is it worth to waste for a $2\times$ throughput improvement?

Chapter 8

Conclusions

This thesis succeeded in its two main goals, as reported in the introduction. On the one hand, it provided an implementation of [10] and [12] which is functional, efficient, and coherent with the expectations. On the other hand, it has built an approximate mathematical framework to solve the problem of throughput improvement when out-of-order execution is allowed in elastic circuits.

The implementation of *Fast Token Delivery* and *Straight to the Queue* is fully integrated into the Dynamatic flow, open source in Dynamatic repository [16] and the results from Chapter 5 can be reproduced. This led to ~ 1.4 time improvement when running circuits, and a ~ 0.8 area improvement. Having a new Pareto-point in the time-area axis is a metric of the success of this work. Unfortunately, the current limitations in the rest of the flow (LSQ, buffering...) do not allow to fully exploit the capabilities of this approach. However, the full integration of *Fast Token Delivery* and *Straight to the Queue* can be part of a future work.

Most importantly, this implementation allows, at a middle-end level, to have all the required steps to work on the out-of-order algorithm from [14].

Starting from such results, the TLP problem was investigated, consisting of letting a circuit elaborate more than one activation at a time, to improve the throughput at which inputs are elaborated. A mathematical framework has been developed, which can state the throughput improvement of moving from a `MUX` to a `MERGE`. This approach can lead to an optimal configuration of `MUX`es and `MERGE`s as loop headers, which allows to correctly introduce out-of-order execution in the circuit.

The results show a 5% error in the estimation for some compiled kernels under analysis. The whole setup is also available on Dynamatic and the results are reproducible.

The methodology, while having promising results, is still at an early stage. The results have been obtained analytically using the semantics of the components. Being a problem of packets flowing into a circuit, it should be handled via an appropriate mathematical framework, such as Network Calculus [21]. This will be the direction of the project. The expectation is that, by the end of the analysis, some results similar to those shown in Chapter 7 will be found.

Bibliography

[1] P. Coussy and A. Morawiec, Eds., *High-Level Synthesis: From Algorithm to Digital Circuit*. Dordrecht: Springer Netherlands, 2008.

[2] "AMD Vitis™ HLS." [Online]. Available: https://www.amd.com/fr/products/software/adaptive-socs-and-fpgas/vitis/vitis-hls.html

[3] "Stratus High-Level Synthesis." [Online]. Available: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html

[4] "Catapult High-Level Synthesis and Verification." [Online]. Available: https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/

[5] L. Josipović, R. Ghosal, and P. Ienne, "Dynamically Scheduled High-level Synthesis," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Monterey CALIFORNIA USA: ACM, Feb. 2018, pp. 127–136.

[6] J. Cortadella, M. Kishinevsky, and B. Grundmann, "Synthesis of synchronous elastic architectures," in *2006 43rd ACM/IEEE Design Automation Conference*, Jul. 2006, pp. 657–662, iSSN: 0738-100X. [Online]. Available: https://ieeexplore.ieee.org/document/1688878

[7] L. Josipovic, A. Bhattacharyya, A. Guerrieri, and P. Ienne, "Shrink It or Shed It! Minimize the Use of LSQs in Dataflow Designs," *2019 International Conference on Field-Programmable Technology (ICFPT)*, pp. 197–205, Dec. 2019, conference Name: 2019 International Conference on Field-Programmable Technology (ICFPT) ISBN: 9781728129433 Place: Tianjin, China Publisher: IEEE.

[8] L. Josipovic, A. Guerrieri, and P. Ienne, "Speculative Dataflow Circuits," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Seaside CA USA: ACM, Feb. 2019, pp. 162–171.

[9] L. Josipović, S. Sheikhha, A. Guerrieri, P. Ienne, and J. Cortadella, "Buffer Placement and Sizing for High-Performance Dataflow Circuits," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '20. New York, NY, USA: Association for Computing Machinery, Feb. 2020, pp. 186–196.

[10] A. Elakhras, A. Guerrieri, L. Josipović, and P. Ienne, "Unleashing parallelism in elastic circuits with faster token delivery," in *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2022, pp. 253–261.

[11] C. Rizzi, A. Guerrieri, P. Ienne, and L. Josipović, "A Comprehensive Timing Model for Accurate Frequency Tuning in Dataflow Circuits," in *2022 32nd International*

Conference on Field-Programmable Logic and Applications (FPL), Aug. 2022, pp. 375–383, iSSN: 1946-1488.

[12] A. Elakhras, R. Sawhney, A. Guerrieri, L. Josipovic, and P. Ienne, "Straight to the Queue: Fast Load-Store Queue Allocation in Dataflow Circuits," in *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey CA USA: ACM, Feb. 2023, pp. 39–45.

[13] J. Liu, M. Graczyk, A. Guerrieri, and L. Josipović, "Fast Switching Activity Estimation for HLS-Produced Dataflow Circuits," in *2024 34th International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2024, pp. 118–125.

[14] A. Elakhras, A. Guerrieri, L. Josipovic, and P. Ienne, "Survival of the Fastest: Enabling More Out-of-Order Execution in Dataflow Circuits," in *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '24. New York, NY, USA: Association for Computing Machinery, Apr. 2024, pp. 44–54.

[15] "Dynamatic Webpage," 2024. [Online]. Available: https://dynamatic.epfl.ch/

[16] "Dynamatic Repository," 2024. [Online]. Available: https://github.com/EPFL-LAP/dynamatic

[17] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Feb. 2021, pp. 2–14.

[18] J. Cheng, J. Wickerson, and G. A. Constantinides, "Dynamic C-Slow Pipelining for HLS," in *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2022, pp. 1–10, iSSN: 2576-2621.

[19] J.-M. Gorius, S. Rokicki, and S. Derrien, "SpecHLS: Speculative Accelerator Design Using High-Level Synthesis," *IEEE Micro*, vol. 42, no. 5, pp. 99–107, Sep. 2022, conference Name: IEEE Micro.

[20] D. Leothaud, J.-M. Gorius, S. Rokicki, and S. Derrien, "Efficient Design Space Exploration for Dynamic & Speculative High-Level Synthesis," in *2024 34th International Conference on Field-Programmable Logic and Applications (FPL)*, Sep. 2024, pp. 109–117, iSSN: 1946-1488.

[21] J.-Y. Le Boudec, P. Thiran, G. Goos, J. Hartmanis, and J. Van Leeuwen, Eds., *Network Calculus*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2001, vol. 2050. [Online]. Available: http://link.springer.com/10.1007/3-540-45318-0

[22] K. D. Cooper and L. Torczon, *Engineering a compiler*. Morgan Kaufmann, 2022.

[23] P. Tu and D. Padua, "Efficient Building and Placing of Gating Functions," *ACM SIGPLAN Notices*, vol. 30, no. 6, pp. 47–55, Jan. 1995.

[24] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, Jul. 1987.

[25] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," in *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium*

on Principles of programming languages, ser. POPL '88. New York, NY, USA: Association for Computing Machinery, Jan. 1988, pp. 12–27.

[26] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, Oct. 1991.

[27] F. Rastello and F. Bouchez Tichadou, Eds., *SSA-based Compiler Design.* Cham: Springer International Publishing, 2022.

[28] K. J. Ottenstein, R. A. Ballance, and A. B. MacCabe, "The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages," in *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, ser. PLDI '90. New York, NY, USA: Association for Computing Machinery, Jun. 1990, pp. 257–271.

[29] P. Havlak, "Construction of thinned gated single-assignment form," in *Languages and Compilers for Parallel Computing*, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds. Berlin, Heidelberg: Springer, 1994.

[30] M. Arenaz, P. Amoedo, and J. Tourino, "Efficiently Building the Gated Single Assignment Form in Codes with Pointers in Modern Optimizing Compilers," in *Euro-Par 2008 – Parallel Processing*, E. Luque, T. Margalef, and D. Benítez, Eds. Berlin, Heidelberg: Springer, 2008, pp. 360–369.

[31] Y. Herklotz, D. Demange, and S. Blazy, "Mechanised Semantics for Gated Static Single Assignment," in *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2023. New York, NY, USA: Association for Computing Machinery, Jan. 2023, pp. 182–196.

[32] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum, "WYSINWYX: What You See Is Not What You eXecute," in *Verified Software: Theories, Tools, Experiments: First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, B. Meyer and J. Woodcock, Eds. Berlin, Heidelberg: Springer, 2008, pp. 202–213.

[33] "MLIR documenation." [Online]. Available: https://mlir.llvm.org/docs/

[34] L. Josipovic, P. Brisk, and P. Ienne, "An Out-of-Order Load-Store Queue for Spatial Computing," *ACM Transactions on Embedded Computing Systems*, vol. 16, no. 5s, pp. 1–19, Oct. 2017.

[35] L. Josipović, A. Guerrieri, and P. Ienne, "Invited Tutorial: Dynamatic: From C/C++ to Dynamically Scheduled Circuits," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '20. New York, NY, USA: Association for Computing Machinery, Feb. 2020, pp. 1–10.

[36] "CIRCT documentation." [Online]. Available: https://circt.llvm.org/

[37] W. S. Moses, L. Chelini, R. Zhao, and O. Zinenko, "Polygeist: Raising C to Polyhedral MLIR," in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2021, pp. 45–59.

[38] L. Josipovic, P. Brisk, and P. Ienne, "From C to elastic circuits," in *2017 51st Asilomar Conference on Signals, Systems, and Computers.* IEEE, 2017, pp. 121–125.

[39] K. D. Cooper, T. J. Harvey, and K. Kennedy, "A simple, fast dominance algorithm," *Software Practice & Experience*, vol. 4, no. 1-10, pp. 1–8, 2001, publisher: Citeseer.

[40] M. Reisinger, "MatthiasJReisinger/PolyBenchC-4.2.1," Mar. 2025, original-date: 2016-06-10T11:45:57Z. [Online]. Available: https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1

[41] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, Nov. 2017.