CGRA configuration generation through schedule based mapping

Brouwer, J.

Award date:
2018

CGRA configuration generation through schedule based mapping

-

Thesis

Justin Brouwer
Student number: 0972541
TU/e

Thursday 2nd August, 2018

Abstract

Coarse-Grained Reconfigurable Architectures (CGRAs) are flexible, energy efficient, high performance computing platforms, and are becoming increasingly more important for actively developed battery powered products, such as mobile sensory processing devices in the medical field.

Automated generation of CGRA configurations to support high performance applications and approximate optimal energy efficiency is challenging. This challenge lies in the immense CGRA design space, and connectivity limitations hindering resource reuse that lead to unpredictable requirements of resources.

This thesis presents a method of clustering instructions based on resource requirements of dependency, and generating CGRA configurations through schedule based mapping of those clusters, enabling subsequent CGRA code generation. This guarantees compliance to a specified schedule while aiming to minimize resource allocation, and thereby energy consumption for a given performance. The proposed method enables developers to generate both the configuration and code for an application, based on a schedule that can be specified to suit the developer's desires. Additionally, schedule based generation is intended to be used in a searching method that approximates the optimal energy efficient implementation by creating different schedules.

The results show that the method guarantees computational performance as specified by the inputted schedule; however, on average it allocates 32% more function units and 37% more connections than expert developed implementations. Before this method can be used in an optimal energy efficiency approximation method, resource allocation needs to be further reduced. Additionally, connectivity limitations pose hindrances on the mapping process, which can result in significant computation time penalties induced by backtracking. These penalties grow exponentially as the backtracking depth increases, and quickly grow to days worth of computation time. Efforts to avoid backtracking through pre-emptive input port reservation, are suggested to be in part responsible for the increase in resource allocation. However, the results also show that the backtracking penalty can be reduced by clustering instructions by their requirements of a shared function unit, and ordering these clusters based on their first executed instruction. Using this order, and limiting the maximum allowed backtracking depth to three mapping steps resulted in an average computation time of 4 seconds per test case, while still successfully completing for 62% of the test cases.

Contents

1. Introduction

Computational energy efficiency is becoming increasingly more important in the world. The demand for mobile, battery powered, high performance computing is ever growing, and finds application in the IoT branch, but also in the medical field[1]. As these are growing fields, the need for updates or upgrades will not be uncommon. To that effect, flexibility is also very important.

Energy efficient high performance computation applications can be implemented by designing a dedicated Application Specific Integrated Circuit (ASIC). However, designing ASICs has large overhead costs and can be very time consuming. Additionally, once ASICs are created it will be difficult to make changes or upgrades to them without having to replace them. This is limiting to its flexibility.

On the other end of the spectrum are Field-Programmable Gate Arrays (FPGAs), which are very flexible while still capable of high performance. As a crude description, FPGAs contain programmable look up tables which are connected through configurable routing resources. This flexibility is made possible using a large number of configuration bits, and is not very energy efficient.

In between those two are the Coarse-Grained Reconfigurable Architectures (CGRAs). These are high performance platforms that are flexible and energy efficient, with some applications in the medical field already[2]. They consist of function units that execute instructions, and communicate using reconfigurable connections.

The flexibility of the CGRA is provided in a similar method as for the FPGA, but in a much more efficient way. Instead of reprogrammable LUTs, the CGRA contains reprogrammable function units, which can be reused for multiple purposes throughout the execution of an application. Additionally as function units only deal with bytes and words, routing in the CGRA configures word or byte sized connections, rather than bit sized as for the FPGA. This reduces the number of configuration bits and provides a more energy efficient solution.

To be able to execute an application on a CGRA, it must be configured with the appropriate function units to execute its instructions, and the appropriate connection paths to satisfy all dependencies. Creating a high performance CGRA configuration with the required function units and connections to execute an application, while maximizing energy efficiency (through balancing function unit reuse and performance increase) is difficult, due to a limit on incoming connections per function unit. This thesis proposes a method to automatically generate CGRA implementations through schedule based mapping, and aims to maximize energy efficiency of the resulting implementations while adhering to the schedule.

First, some background information on CGRAs and related work is discussed in chapter 2. The proposed method is part of an overarching tool flow that searches for energy efficient implementations for provided applications, which is discussed in chapter 3. This tool flow consists of a scheduler and an implementation generator. The scheduler, which is discussed in chapter 5, searches for a schedule that results in the most energy efficient implementation. The implementation generator generates configurations through mapping based on a provided schedule, which is the main topic of research of this thesis and discussed in chapter 4. Chapters 6 and 7 discuss the experiments and results respectively, which determine whether the proposed method has merit. Chapter 8 discusses the conclusions derived from the research. Chapter 9 concludes this thesis, and discusses possible improvements to the proposed method.

1.1 Contributions

This thesis proposes a method to generate configurations and code for CGRAs through mapping based on predetermined schedules. The mapping process generates a configuration that guarantees the related application is executed exactly conforming the provided schedule. The resulting mapping itself can be used to generate the code for the application that will run on the CGRA.

It also describes how dependencies can be classified and instructions can be clustered, based on the requirements of the dependencies. This enables simultaneous mapping of instruction clusters, rather than individual instructions.

2. Related Work

Many different CGRAs have been created with unique features and capabilities[5]. Despite this, uniform development is still quite difficult, as many implementations slightly differ from each other. Therefore, this thesis will focus on the Blocks CGRA, which is developed by Technische Universiteit Eindhoven (TU/e)[4].

This chapter aims to provide an insight into what is already done for Blocks, which will be discussed in section 2.1. It will discuss the design space of the CGRAs and its relation to energy consumption in section 2.2. Additionally, it will discuss what techniques of generating architectures or configurations could possibly be of interest in section 2.3.

2.1 Blocks

A Blocks implementation requires both a configuration and parallel assembler code. Figure 2.1 shows a schematic overview of a Blocks CGRA. It shows the building blocks that make up a configuration, and how they relate to each other. Configurations contain no physical information, such as placing and routing. The instruction memory contains the parallel assembler code.

Function units and the connections between them are discussed in subsection 2.1.1. Subsection 2.1.2 discusses the instruction feeds that provide function units with their instructions. Memory is discussed in subsection 2.1.3. Parallel assembler code is discussed in subsection 2.1.4. Subsection 2.1.5 reiterates the limiting factors that complicate CGRA design. Subsection 2.1.6 discusses some of the work that has been done for Blocks already. The section is concluded by subsection 2.1.7, which discusses what is yet missing from the Blocks tools.

2.1.1 Function units & connections

Function units (FUs) are the work-horse of the CGRA. They are the modules that load, compute and store all data handled by the CGRA. Each individual FU is responsible for a limited set of tasks, such as multiplications, arithmetic, load/store access to memory, etc. Each FU type has a dedicated instruction set specific to its purpose.

At the time of writing, the available FU types are:

- Accumulate & branch unit (ABU). Used as either a multi-register accumulator, or as a branching unit.
- Arithmetic logic unit (ALU). Used for simple arithmetic: additions, subtractions, comparisons, bitwise logic operations (OR, XOR, AND, etc.), and logic and arithmetic shifts. Can be configured for unbuffered computations (e.g. 0 cycle computation, without storage).
- Immediate unit (IU). Used to input constants through instructions.
- Load store unit (LSU). Used to store and load from the global and local memory. Has 16 registers used for implicit addressing. Can load and store simultaneously. Loading from and writing to global memory causes stalled cycles.
- Multiplier unit (MUL). Used for multiplications.
- Register file (RF). Used for quick temporary storage. Has 16 registers. Load results are available within the same cycle as execution. Can load and store simultaneously.

Figure 2.2 shows a generic example of a FU. Nearly all FU types have four input ports and two output ports. The immediate unit is an exception, as it has no input ports and only one output

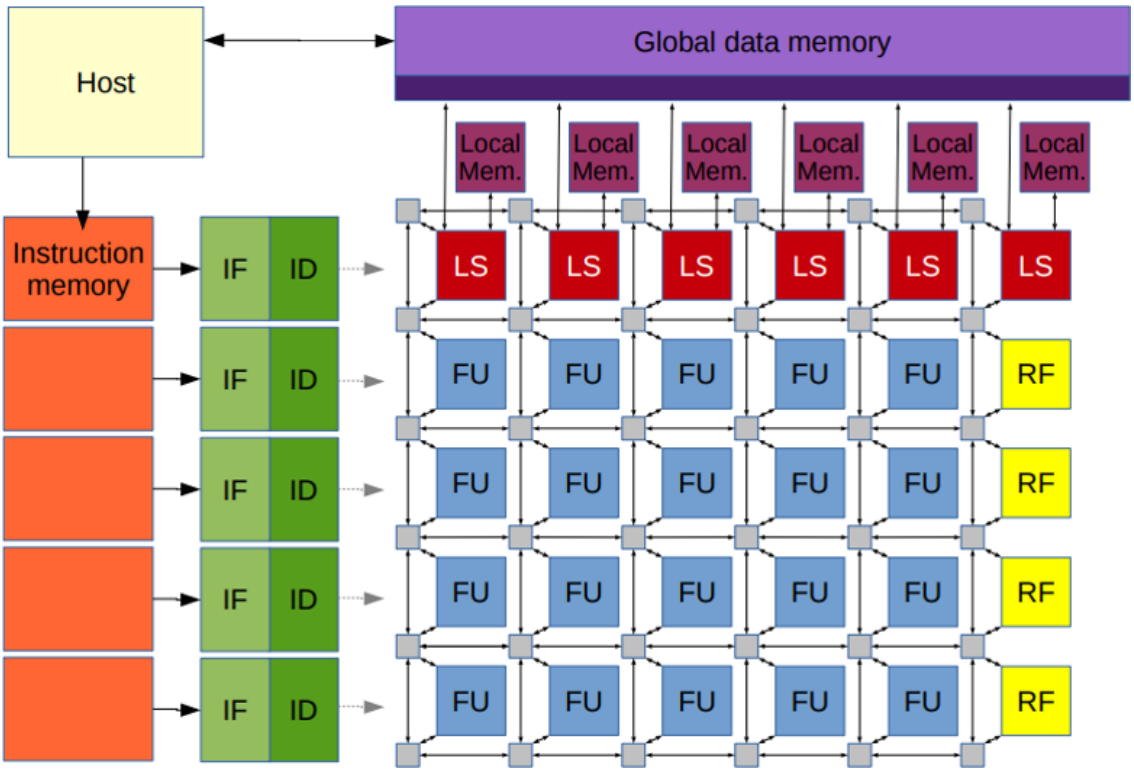


Figure 2.1: A generalized overview of the CGRA. Image from TU/e Embedded Computer Architecture course (5SIA0 2016-2017).

port. Most FU types have output registers connected to their output ports, such that values produced by the FU can be buffered. Some FU types, like the unbuffered ALU, have one or two output ports without output register. On these output ports, values are only available during the cycle they are created.
A number of FU types have internal registers. For instance, the register file and load store unit both have internal registers, but use them differently.

Some function units have configurable behavior. For instance, the ABU can be used as an accumulator, but also as the FU that controls the program counter (i.e. the branch unit).

Each FU can execute one instruction per cycle. As CGRAs consist of multiple FUs, this results in a setup similar to VLIW CPUs, in that each FU can execute a unique instruction simultaneously with other FUs. Nearly all instructions finish execution within one cycle, except for the branching instructions, which have two delay slots.

The FUs are connected to each other through their output and input ports. The schematic overview in figure 2.1 shows that each FU is connected to gray nodes. These gray nodes are routing hubs, and are configured to create dedicated connection paths between the ports of FUs. These hubs allow the connections of the CGRA to be reconfigured. It is also possible for the CGRA to use static connections, which are not reconfigurable and would reduce its flexibility.

Each input port can read data from only one output port; however, multiple input ports can read from the same output port. As there are only four input ports per FU, this means that each FU can read from at most four unique output ports.

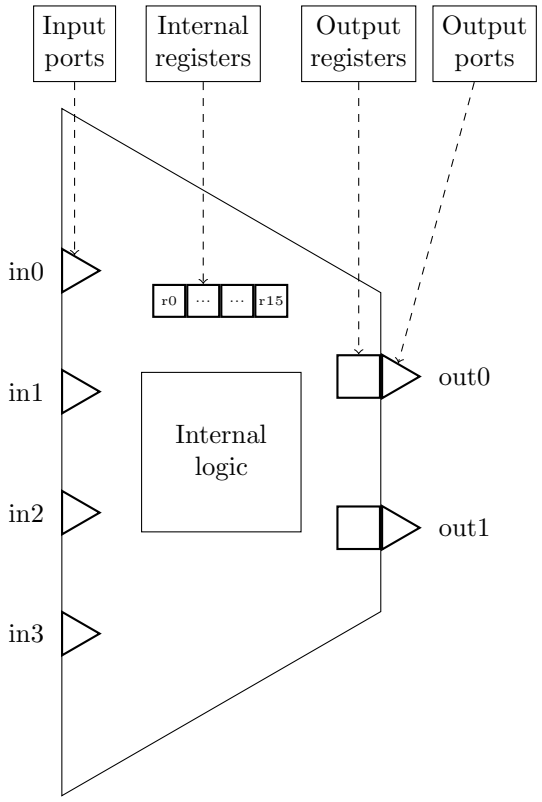


Figure 2.2: Generic FU

2.1.2 Instruction feeds

Instruction feeds load instructions from instruction memory and provide them to FUs. The connections between instruction feeds and FUs are similar to the connections between FUs and are also reconfigurable.

Figure 2.3 shows an example of how instruction feeds can be connected to FUs. For simplicity, memory is left out of the figure. The red arrows are the connections between instruction feeds and FUs. As can be seen from the figure, instruction feeds can supply instructions to multiple FUs simultaneously. This is similar to SIMD techniques, as multiple FUs will execute same instruction simultaneously. This will be referred to as instruction feed sharing, and can be used to increase energy efficiency.

Instruction feeds are controlled by a single ABU that is configured as a branching unit. Controlling separate sets of instruction feeds with multiple ALUs creates a situation similar to multi-core CPUs. Both instruction feed sharing and multi-core implementations are out of scope for this project. Despite their clear benefits, they are too complicated to take into account for this first step of our proposed method.

2.1.3 Memory

Blocks contains two types of memory: global memory and local memory.
Global memory is accessible through all LSUs. It contains the data that is provided by the host, and is where the output of a CGRA application is written. Access to the global memory causes stalled cycles due to the shared bus access.
Local memory is dedicated to individual LSUs. Each LSU has access to its own local memory,

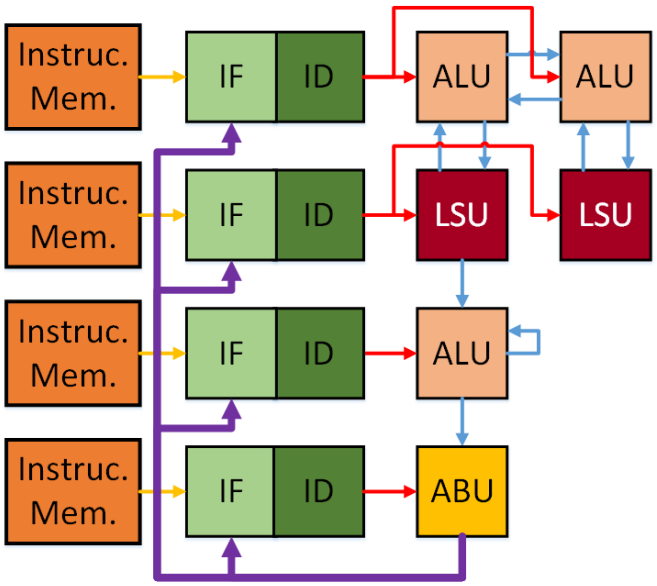


Figure 2.3: Example of instruction feeds & connectivity

which can not be accessed through other LSUs. Accessing this memory causes no stalled cycles, but can only be used for temporary storage.

2.1.4 Parallel Assembler

Parallel Assembler (PASM) is the assembler for Blocks implementations. Listing 2.1 shows an example of a small application that loads two words from global memory, sums them and stores the result at another address in global memory. Each column contains instructions for the instruction feed indicated at the top row. Instructions generally indicate how they relate to the configuration, by specifying output ports, input ports, and registers. However, instructions are not always explicit about which input port or output port they use (e.g. 'imm 245' in the IU column writes to out0).

```
| IF_ABU      | IF_LSU              | IF_ALU                  | IU          |
|             |                     |                         |             |
| nop         | nop                 | nop                     | imm 245     |
| nop         | srm r9, in0         | nop                     | imm 1       |
| nop         | srm r2, in0         | nop                     | nopi        |
| nop         | lgi WORD, out0      | nop                     | nopi        |
| nop         | lgi WORD, out1      | nop                     | imm 16564   |
| nop         | srm r13, in0        | add out0, in0, in1      | nopi        |
| nop         | sgi WORD, in1       | nop                     | nopi        |
| jai 0       | nop                 | nop                     | nopi        |
```

Listing 2.1: PASM example code

2.1.5 Important Limiting Factors

There are two main limiting factors in CGRA configurations, which cause some complexity for development.

The first is that FUs have a limited number of input ports. Each FU can receive data from only four unique output ports. This can severely limit reusability of a FU.

The second is that FUs have a limited number of output registers. While a produced value is still depended upon, it should not be overwritten. If both output registers of a FU contain values

that are depended on, then the FU should not execute instructions that produce new values. Using pass instructions (or store and load instructions on a register file), the values can be moved. However, they should always be moved such that it is still possible for the depending instructions to read the value.

2.1.6 Previous work

Some work has already been done for Blocks, and several tools are already available for use.

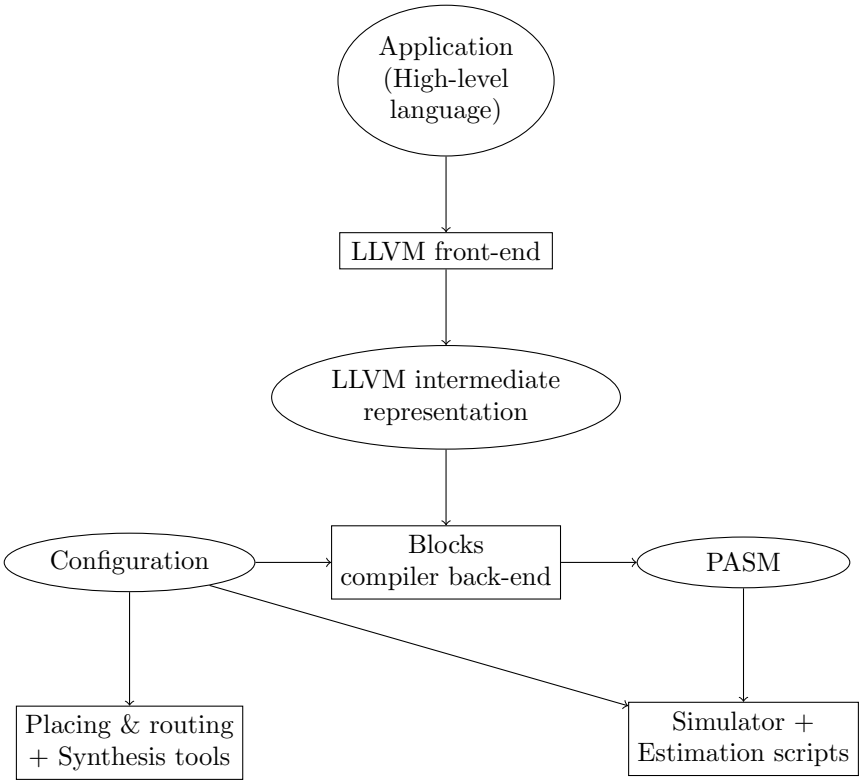


Figure 2.4: Overview of the current toolchain

Figure 2.4 shows the tools that are currently available, and which products they create. Below, some of those tools are discussed as well as other work that was done for Blocks.

PASM Compiler

The Blocks compiler back-end converts LLVM intermediate representation into target specific instructions, and performs mapping and scheduling[7]. It also requires a CGRA configuration as input.

Simulator + Estimation scripts

The simulator is based on ModelSim. It requires both a configuration and PASM as input. It runs a model of the CGRA and can accurately determine execution time in cycles, and utilization numbers. Additional scripts provide estimations for area, power, and energy consumption.

Placing & routing + Synthesis tools

The configuration can be used for placing and routing. The result can then be used to synthesize a hardware design of the CGRA implementation.

Optimal Instruction Scheduler

The optimal instruction scheduler is a constraint solver which can find a performance-wise optimal schedule[11]. It requires assembler level instructions and their dependencies, and a configuration as input. It has a long computation time and at the time of writing only exists as a model.

Loop overhead reduction techniques

Research done by [3] has provided the Blocks CGRA with some interesting upgrades, such as the zero-overhead loop accelerator (ZOLA) and loop buffers. These reduce the instruction memory activity of the CGRA while iterating through a for-loop, by either repeating an instruction without reloading (for single cycle loop nests), or by buffering the instructions in the for-loop. This decreases energy consumption significantly, especially for very short loop-nests resulting in high performance applications.

2.1.7 What is missing

Figure 2.4 shows that all of the Blocks specific tools require a configuration to work; however, there is no tool to create one. All configurations are currently created by hand.

Although the back-end compiler takes care of a lot of the development work, the developer still needs to accurately estimate what is required of the configuration to achieve good results using the back-end compiler. Lacking a specific connection or FU can hinder the mapping process and result in a poor schedule, if it succeeds at all.

Crude estimations of the amount and types of FUs and connections can be made. These estimations can be based on the instructions and dependencies in the application, and the desired level of parallelism. However, these are far from perfect and do not guarantee an intended performance level is achieved. When using these tools to search for an energy efficient implementation of an application, the developer would need to iteratively check multiple configurations to find which result is best.

Clearly, this flow could benefit from automatic generation of configurations. This could reduce development time. Additionally, it could allow for a better design space exploration and result in a more energy efficient solution.

2.2 The magnitude of the design space

The CGRA has a rather large design space, and exhaustive search for an energy efficient configuration to match an application is a big challenge[17]. This section will attempt to illustrate the magnitude of the design space, by discussing some of the myriad of considerations and options. The focus will be on design choices regarding energy efficiency.

2.2.1 Bypassing

The ability to directly connect FUs to each other has significant benefits. Any value produced by a FU can be used directly in the next cycle, rather than having to be written back to and loaded from a register file, like in many CPUs. This is called bypassing. As shown by [6], this can save energy that would otherwise be spent on performing register file read and write operations. Bypassing can be implemented in CGRAs and has positive effects on performance and energy consumption.

However, one of the benefits of the register file in the CPU is that is serves as a data-hub. All modules that require data that is not spilled to memory, can retrieve it from the register file.

Due to the input port limitation of the CGRA, each FU can only directly receive data from four unique output ports. Bypassing is therefore limited, and care has to be given which FU connects to which FU, to assure they can receive required data. The configuration might benefit from such a data-hub.

2.2.2 Register Usage

FUs have a limited number of output registers. As described in 2.1.5, this could cause some issues. A FU should not overwrite live values (i.e. values that are still depended upon) in its output registers. If all output registers of a FU contain live values, the FU is effectively blocked from performing operations that write values to output registers, until a value is moved or 'dies'. There are several methods to deal with such a situation.

Different scheduling of instructions

Instructions can occasionally be rescheduled in such a way that the lifetime of the blocking values and the blocked instruction do not overlap. This can negatively affect runtime, but would free up a register for other values to be stored in. Sometimes, rescheduling requires allocating additional FUs.

Different mapping of the instructions

Instructions can be mapped to different FUs, such that the lifetime of values do not overlap on the same register. Figure 2.5 depicts values mapped to output registers of two FUs. Their lifespans are indicated with red rectangles. As depicted by figure 2.6, remapping the values to different registers allows a single registers to become consistently available for a longer period of time. This register can subsequently be used by a value with a long lifespan, marked in green. Remapping can require allocation of additional FUs.

This remapping can be achieved through graph coloring[8]. One can possibly also add preferences based on how the consumers of the live-values are connected to the output registers, using a somewhat different method[9]. This can be beneficial to minimizing the number of required connections.

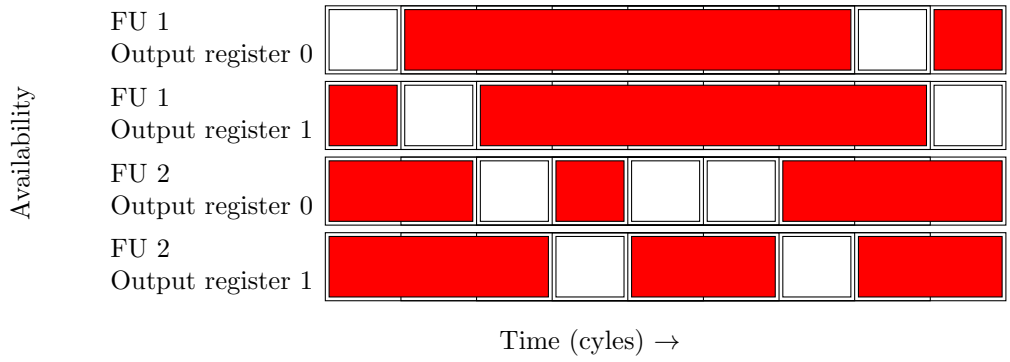


Figure 2.5: Inefficient mapping of live values; Red: output register contains live value; White: output register is available

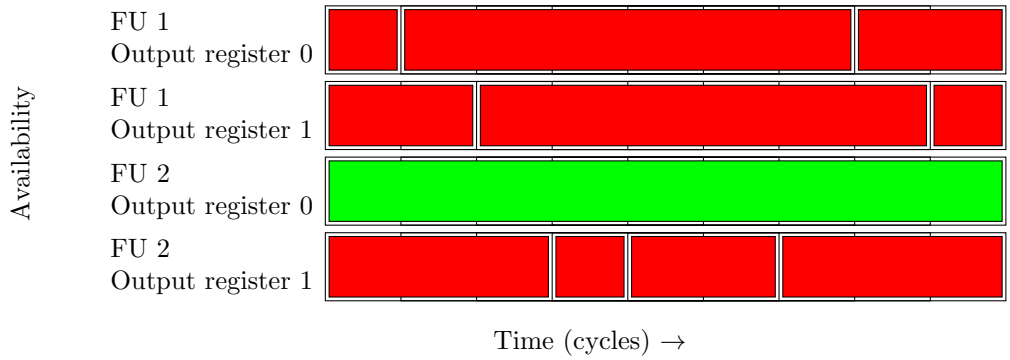


Figure 2.6: Efficient mapping of live values; Red: output register contains live value; Green: reserved for new live value

Temporary storage

Another solution is to move and temporarily store values elsewhere. A register file can be used for this. However, other FUs that also have registers could serve as temporary storage as well. Such a temporary storage FU might need to be allocated.

Most FUs have two output registers, RFs have 16 internal registers, and LSUs have access to local and global memory. However, this increased capacity often means more area is used. Therefore, a simple ALU might sometimes be the optimal solution when considering energy consumption. Additionally, accessing global memory produces stalled cycles which can negatively impact performance.

2.2.3 Parallelism exploitation

CGRA configurations can consist of large number of function units, in various combinations. This enables easy exploitation of parallelism. Parallelism exploitation is highly recommended, as long as FUs are sufficiently utilized.

More FUs often means more work can be done per cycle, which can reduce the runtime (in cycles) significantly. Of course, this fully depends on how well the application can be parallelized. Additionally, which FUs and connections are required to improve runtime is not always predictable. On top of that, there is a trade-off. Reducing runtime has a positive effect on energy consumption, while the additional resources have a negative effect. Whichever is dominant is not always predictable and largely depends on the application.

2.2.4 Shared instruction feeds

Sharing instruction feeds can save energy through parallelism exploitation, without increasing instruction memory activity[6]. Whether instruction feeds are shared is specified in the configuration, and requires changes in the PASM code. This should only be applied to FUs that perform the same function throughout the entire application.

2.2.5 Loop Nest Pipelining

Loop nest pipelining is a specific case of parallelism exploitation. It enables instructions that need to be executed sequentially in loop-nests, to be executed in parallel. This is achieved by simultaneously executing instructions from different iterations of the for loop. In effect, the instructions of a single iteration (of the original for loop) are still executed sequentially, but in parallel to

instructions from other iterations. Whether this is possible depends on inter-iteration dependencies. Whether loop nest pipelining is beneficial to energy consumption depends on its impact on runtime and how many resources needs to be allocated to achieve it.

The Blocks CGRA employs some techniques that can reduce instruction memory activity for loop nests of limited length[3], which reduces energy consumption. These techniques can benefit from pipelining.

2.3 Generation techniques

This section discusses some possible generation techniques for CGRA configurations. Some are inspired by fields such as transport triggered architectures (TTAs).

2.3.1 Pareto Curve search

This method originates from a synthesis tool for transport triggered processors[12]. The method attempts to approximate a Pareto curve while traversing through the design space. This efficiently explores the design space while yielding potentially interesting results. Each Pareto point represents an architecture or configuration, and compares execution time with resource costs (e.g. area, or total number of resources, etc.). This could perhaps be used for energy efficiency as well.

The method starts at an extreme point of the Pareto curve. For the TTA, this extreme point is a configuration with a fully connected set of FUs (i.e. each FU connects to each FU). It then iteratively considers minor changes to the configuration by removing resources.

By assigning a quality value to each configuration, the benefit of these changes can be quantified. The quality is based on the execution time and resource cost, as shown in equation 2.1, where α is the weight for resource costs, and β is the weight for execution time.

$$quality(config) = \frac{1}{costs(config)^{\alpha} * execution\_time(config)^{\beta}} \tag{2.1}$$

By applying weights to both execution time and resource costs, the preference of the developer can be translated into the calculation of the quality value. This in turn forces the searching method to consider different paths while attempting to traverse the Pareto curve with different weights.

Unfortunately, this method is not very suitable for CGRA configuration generation. The searching method requires an extreme starting point, either a 'minimal configuration' or a 'maximum configuration'. Both are not easy to obtain. The TTA setup allows for a fully connected configuration in which all FUs connect to all FUs. This is used as the 'maximum configuration' starting point. However, due to the input ports limitation, this is not possible for the CGRA.

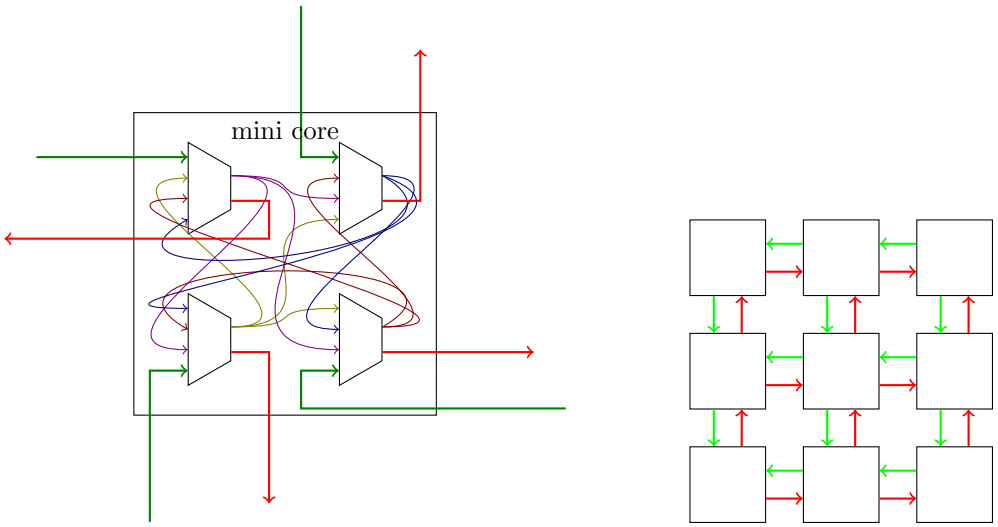
Additionally, due to the limited connectivity, progression through the design space could lead to local minima if connections are only allowed to be removed or added. Only adding connections would make some added connections obsolete, while their reserved input ports could be better used for other connections.

2.3.2 Mini Core Solution

The mini cores solution can reduce the complexity of CGRA configuration generation by using mini cores as building blocks, instead of individual FUs[14]. These mini cores are small clusters of connected FUs. This method mitigates the problems caused by the input port limitation and register limitation.

As illustrated by figure 2.7a, three of the four input ports of a FU would connect to the other FUs within the same core. A total of four input ports remain available to connect to other mini cores. If set up like a grid, as depicted in figure 2.7b, this allows all FUs to (indirectly) reach all other FUs. This allows optimal use of the available registers, and solve input port limitations through indirect connections. Additionally, it also benefits the flexibility of the configuration. However, this would severely impact performance as each communication hop through other FUs requires an instruction to pass data.

Mini cores are not very suitable for approximating the energy-wise optimal configuration for



(a) Mini core (b) Mini core grid setup

Figure 2.7: Mini core examples

an application. The restriction put on the connections can force data to travel indirect routes, which negatively influence performance. Additionally, such indirect data communication utilizes more FUs and connections than communication through direct connections. To compensate, additional resources could be allocated leading to more area, which can be a dominating factor in energy consumption on some platforms[16].

An additional issue with this method is tool support. The necessity for passing data through FUs will increase the complexity of the compiler. It is a complex task to determine when and how many data-passing instructions are needed and/or beneficial, and for which dependencies.

2.3.3 Template Library Solution

The template library solution attempts to provide larger granularity building blocks to support a wide range of applications.
After analysis of a provided application, specific building blocks will be loaded from a template library and combined to match the profile of the application. This technique is applied in ASIC design [20], and could possibly be applied for CGRA design as well.

Standardized sub-configurations can be designed for specific operations or more general applications, such as matrix multiplication or signal filtering. The strength of this solution lies in the reduced time-to-market.

Quite a few templates need to be created to support sufficient applications. Granularity plays

a significant role in this. Designing the templates to be robust and flexible, allows a large range of applications. However, this is would come at the price of performance, area, and/or energy efficiency. Reducing the granularity, to allow for more focus on performance, area, and/or energy efficiency, would yet increase the complexity of allocating template building blocks and properly connecting them.

2.3.4 Expand the optimal instruction scheduler

This solution would build on A. Tiemersma's work on the Blocks optimal instruction scheduler[11]. This scheduler is capable of creating a performance-wise optimal schedule, based on a configuration and an application.

The scheduler is a constraint solver which maps the application to the provided configuration, and in that way producing a schedule. By defining a cost function based on the schedule length, it attempts to reduce runtime as much as possible.

The constraint solver could be expanded to include configuration generation as well. However, it has shown that scheduling time can be rather unpredictable and sometimes have significant consequences. As concluded by [11], providing a configuration with sufficient parallelism significantly reduced the scheduling time (e.g. one of the results was reduced from 4135 seconds of scheduling time to 0.7 seconds by adding a single FU). However, approximating energy efficient results, rather than performance optimal results, may include efficient reuse of resources, instead of configurations supporting parallelism.

2.3.5 Generation based on Statistical Analysis

Configuration generation for CGRA, based on statistical analysis is inspired by a similar approach applied by R. Jordans[15] for instruction-set architecture synthesis for VLIW processors. As the CGRA can be quite similar to VLIW processors, this method can possibly be altered to suit the CGRA platform as well.

The method first analyzes a provided application to determine possible opportunities for parallelism exploitation, and discover sections that have a large work load.

Based on this analysis, the application can be restructured. For the CGRA this would entail scheduling, and reducing register usage. This would serve the purpose of maximizing performance while minimizing resource requirements. Instruction level parallelism estimations provide a crude estimation of how many function units are minimally required in a configuration, and can be used in this process of restructuring. How often sections are executed is also desirable information. Based on the resulting application, an initial configuration can be generated.

Finally, the configuration would be refined further, until the application is perfectly supported.

3. Schedule based search

The related works chapter indicates that a method for configuration generation is missing in the current tools (subsection 2.1.7). Such a method could be used to supplement the implied searching method to find energy efficient implementations for applications.

The implied searching method first generates a configuration, performs mapping and scheduling in the compiler back-end, and retries this to approximate the optimal energy efficient solution. However, the schedule resulting from the compiler back-end can only be as 'fast' as the configuration allows. The configuration generation stage of the searching method would dictate area, power consumption, and indirectly put a minimum on the makespan of the schedule (e.g. a single ALU can only do half the work of two ALUs in the same timespan). Through these metrics, it would also influence the potential minimum energy consumption.

Attempting to control energy consumption through configuration generation is quite complex and unpredictable. Especially the effects on performance can be quite unexpected. Due to the input port limitation, additional FUs might not always be properly connectable, and might not have the intended effect on scheduling. Additionally, finding an optimal schedule for a configuration does not mean that the configuration is fully utilized, which would mean energy is still wasted.

This chapter proposes a different searching method. This new method attempts to approximate the optimal energy efficient solution by creating various schedules and generating minimal resource configurations that support them. By iteratively changing the created schedules, the searching method can gradually approximate the optimal energy efficient solution. In the context of this proposed searching method, a method is proposed to generate configurations through schedule based mapping. This method of configuration generation is the main focus of this thesis, and will be discussed in chapter 4.

The proposed searching method requires a new tool flow, which is explained in section 3.1. After that, section 3.2 provides some background information on how the CGRA can affect energy consumption, and how that relates to our tool flow. Although the method will seem counter intuitive, it does have a number of benefits. These are discussed in section 3.3.

3.1 The proposed tool flow

The proposed tool flow consists of a number of tools. This section aims to give a small overview of those tools. Some tools are elaborated on in other chapters, which will be referred to in their respective subsections.

Figure 3.1 shows an overview of the proposed tool flow. First, a schedule is created based on the application created by the developer. The scheduler is discussed in subsection 3.1.1.

The resulting schedule is fed to the implementation generator tool, which is discussed in subsection 3.1.2. This is the proposed configuration generation method, which coincidentally also is capable of generating PASM. The configuration and PASM serve as input for the existing Blocks tools.

Estimations of energy and other metrics are made based on the configuration and PASM. This is discussed in subsection 3.1.3. The estimations can then be used by the scheduler as feedback, to make small alterations to the schedule. Repeating this cycle over and over allows the scheduler to approximate the optimal energy efficient schedule.

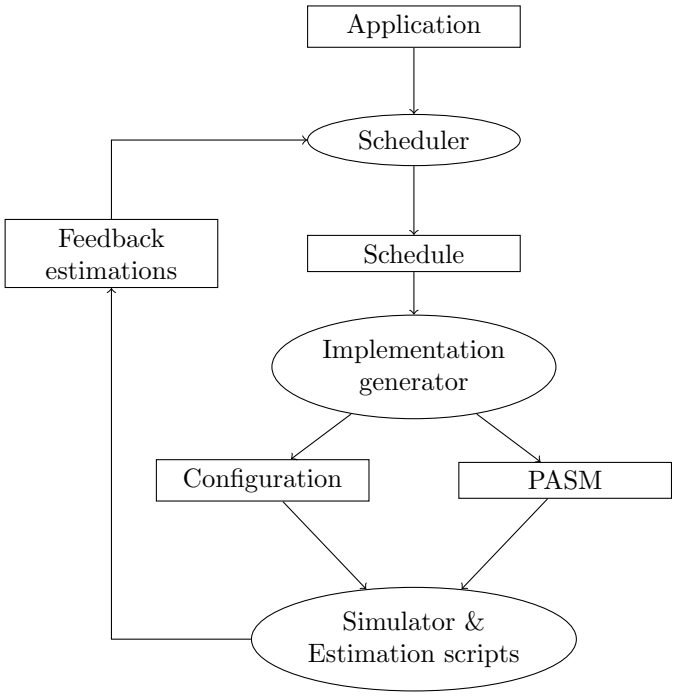


Figure 3.1: Overview of the proposed tool flow

This tool flow is inspired by the generation method as discussed in subsection 2.3.5. The scheduler is responsible for application analysis and restructuring, and the implementation generator is responsible for generating a configuration and PASM for the resulting schedule. The final step, which entails further refinement of the configuration, is not covered in this thesis.

3.1.1 Scheduler

The scheduler is not only responsible for scheduling, but also for searching for an energy efficient solution. As will be explained in section 3.2, the optimal energy efficiency implementation is expected to be a high performance implementation. Therefore, the ability to create implementations according to specified performance can be beneficial for the search.

During the initial run of the scheduler, the goal is to produce a specific high performance implementation through parallelism exploitation. This will lead to a schedule with ASAP scheduled basic blocks and (where possible) pipelined loop nests. It can be possible to increase performance further by duplicating pipelined loop nest contents, but this will not be applied for the initial schedule. This schedule is expected to be relatively cheap to compute (computation time-wise).

In subsequent runs, the scheduler makes small alterations to either reduce the requirements posed on the configuration, or increase performance. Scheduling some instructions sequentially can reduce the requirements on the configuration, while performance can be increased by scheduling instructions in parallel. Based on FU utilization numbers, basic block execution numbers, and collected information about the CDFG, it can make smart choices regarding which instructions and basic blocks to target.

The scheduler is further elaborated on in chapter 5.

3.1.2 Implementation generator

The goal of the proposed implementation generator is to generate a minimum resource configuration and PASM code that is scheduled exactly according to the provided schedule.

By mapping instructions to configuration resources, the generator discovers which resources are lacking to complete the mapping. This mapping also entails mapping dependencies to connections and produced values to registers.

Traditional mapping tools often adjust the schedule when failing to map an instruction[21][13]. The proposed implementation generator breaks with tradition and allocates the missing resources instead. In this manner, the configuration 'grows' resources to provide support for the schedule.

Mapping the schedule onto the configuration asserts that the provided schedule is supported, and that no resource is allocated that remains unutilized. A more in depth discussion of this process will be presented in chapter 4.

3.1.3 Simulator & Estimation scripts

The phase following implementation generation collects feedback for the scheduler. The scheduler can use this feedback to make small alterations to the schedule and try a different one. The metrics that are expected to be of interest are runtime, number and type of resources (contained in the configuration), resource utilization (both FUs and connections), basic block execution numbers, energy consumption, power dissipation, and possibly area.

Runtime, utilization, and functional correctness can be determined by the simulator. At the time of writing, estimation scripts exist for energy consumption, power dissipation, and area. Basic block execution numbers can be discovered through minor alterations to the schedule, but would preferably be determined by the simulator.

3.2 Energy efficient implementation

This section discusses how energy consumption can be influenced by the CGRA.

$$E_{total} = E_{static} + E_{dynamic} \tag{3.1}$$

Equation 3.1 shows the two main terms which together make up the energy consumption of an electric circuit, where E_{total} is total energy, E_{static} is static energy and $E_{dynamic}$ is dynamic energy. How these terms can be influenced by the CGRA is discussed in subsections 3.2.1 and 3.2.2 respectively. From the concluding discussion in subsection 3.2.3, we will learn how our proposed tool flow can contribute to reducing energy consumption.

3.2.1 Dynamic energy consumption

Dynamic energy is the energy consumed through the switching of transistors. One of the methods to calculate it is shown in equation 3.2, where $E_{dynamic}$ is the dynamic energy, α_{total} is the total switching activity and E_{switch} is the average energy consumed for a single switch.Switching activity refers to transistor value switches, and total switching activity refers to the total number of switches during execution.

$$E_{dynamic} = \alpha_{total} * E_{switch} \tag{3.2}$$

Total switching activity is predominantly determined by the application. There are some tricks to reduce total switching activity, which deal with code optimizations. It could therefore be beneficial

to include some code optimization in the scheduler as well.

Sharing instruction feeds is a method to reduce switching activity in instruction memory. This poses complex requirements on both the scheduler and especially the implementation generator. At this stage of development, this would be too complex to incorporate, and is therefore out of scope.

Switches consume energy. The energy required for an individual switch depends on physical implementation specifics, such as voltage levels, and what kind of technology is used (e.g. different transistor lengths).

Voltage levels can be influenced indirectly by the scheduler. Voltage scaling enables voltage to be reduced at the price of a longer clock cycle length. The scheduler can compensate for this by reducing runtime in cycles[22].
The formula for dynamic power in CMOS circuits is presented in equation 3.3, where C is load capacitance, V is voltage level, $\alpha_{average}$ is the average switching activity in a cycle, and f is the clock frequency. It shows that voltage is a dominant term in dynamic power dissipation. Voltage scaling can reduce power squared, while the clock cycle length increases linearly. This can be beneficial to energy consumption.

$$P_{dynamic} = \frac{1}{2} * C * V^2 * \alpha_{cycle} * f \tag{3.3}$$

If the scheduler is capable of reducing clock cycles as much as possible, voltage scaling could be applied to save energy. This encourages high performance schedules.

3.2.2 Static energy consumption

Equation 3.4 shows how static energy can be calculated, where E_{static} is static energy, $t_{runtime}$ is runtime, and P_{static} is static power dissipation.

$$E_{static} = t_{runtime} * P_{static} \tag{3.4}$$

Static power dissipation is also referred to as static leakage. It is the power that dissipates simply by powering transistors. It originates from subthreshold leakage. How much static power dissipates depends on transistor technology (e.g. low threshold vs high threshold), voltage level, and the amount of transistors (that are powered).

FUs, instruction feeds, and dynamic connections all contain transistors. The configuration determines how many of these resources are needed, and therefore has influence on the number of transistors. Note that the number of transistors per FU type can differ.

Runtime can be reduced by reducing clock cycle length, or reducing the number of clock cycles. Clock cycle length can be reduced i.a. through voltage scaling. However, as discussed in subsection 3.2.1, increasing voltage has a squared effect on dynamic power which could outweigh the reduction in runtime.

Reducing the number of clock cycles can be achieved by exploiting parallelism, but could require additional resources and therefore additional static power dissipation. Which effect is dominant depends on the change in runtime and static power dissipation.

3.2.3 Discussion

This subsection discusses how the proposed tool flow can exploit the influences on energy consumption.

The scheduler

The scheduler searches for the schedule which results in as little energy consumption as possible. By setting the length of basic blocks, the scheduler directly influences the runtime of the final implementation, and therefore static energy consumption. This can also facilitate voltage scaling to reduce dynamic energy consumption. Being able to directly control the resulting performance of the implementation is one of the strengths of the proposed searching method.

The scheduler might also be able to apply code optimizations (e.g. to increase data reuse), such that signal activity can be reduced. This influences dynamic energy consumption.

The scheduler has indirect influence on the potential minimum of static power dissipation. The created schedule poses requirements on the configuration, which dictates a minimum on the number of resources of each type (e.g. executing five multiplications simultaneously requires at least five multipliers in the configuration).

The scheduler can therefore influence the trade off between runtime and static power dissipation. Adding resources has a negative effect on static energy consumption, while reducing basic block lengths has a positive effect. The scheduler predicts which effect is dominant and attempts reduce energy consumption using those predictions. Changes do not always have the predicted effect, and the scheduler will therefore repeatedly try different changes. Basic blocks that are executed often, are expected to be more likely to have a positive effect on energy consumption if parallelized.

Reducing the total number of cycles is expected to be the dominant factor in energy consumption reduction. It can have positive effects on both dynamic and static energy consumption. Therefore, the scheduler will initiate its search with a high performance schedule.

The implementation generator

The implementation generator is responsible for generating a configuration and PASM code that support the provided schedule exactly. This constraint simplifies the goal of the implementation generator. Exactly supporting the schedule bars the implementation generator affecting runtime, instruction level parallelism, or data reuse.

Static power dissipation depends on the number **and** type of resources that are allocated. However, each resource type provides unique functionalities and rarely are able to replace each other. There are some cases, such as temporary storage of values, which can be fulfilled by the same FU. However, these cases require changes to the schedule, which is not allowed.

In the future it might be beneficial to allow changes to the schedule that do not affect basic block length. This could increase the computation time of the implementation generator, and reduce the search space of the tool flow. However, the implementation generator as-is will only need to focus on reducing the number of resources per resource type.

3.3 Benefits

The most significant expected benefits of the proposed tool flow are discussed in this section.

3.3.1 Matching configurations

One of the strengths of the proposed tool flow lies in matching the configuration to the requirements of a schedule.

In a traditional tool flow, the schedule would follow from mapping an application onto an already

generated configuration (or platform). The resulting schedule would only be as good as the configuration can support. This is beneficial when the developer is already bound to an existing configuration.

However, it might turn out that the configuration is not a proper match. Some FUs or connections might not contribute anything at all to the application, and would waste energy. Inversely, lacking a single FU or connection could limit mapping and result in a poor schedule (and in turn could also leave resources unused). It might even turn out that mapping the application has become impossible.

The tool flow aims to find the best possible schedule, and finds the best fitting configuration to support it. In this case, the best configuration is the one with the fewest resources. Due to the proposed configuration generation method, the configuration should have no unused FUs, nor unused connections. This is quite beneficial when the developer is not bound to a specific platform. Our proposed tool flow attempts to approximate the theoretical optimal energy efficient solution. The developer can then select (e.g. buy, synthesize, etc.) a fitting platform that supports it (i.e. a platform which has few, or no resources in excess).

If the developer is bound to a specific platform (e.g. when updating or upgrading an existing product that already contains a CGRA platform), the proposed tool flow can still be applied. By specifying restrictions for the implementation generator, based on the number and type of allowed resources, the proposed tool flow will eventually find a fitting configuration by adjusting the schedule to match.

3.3.2 Dictating performance

The counter intuitive decision of starting with scheduling has the benefit that performance can easily be dictated. Section 3.2 suggests that runtime is a dominating influence on energy consumption. Additionally, it also suggests that the optimal energy efficient implementation might be a high performance solution. By being able to dictate performance (rather than it being an emerging property, like in traditional tool flows), the searching method can start its search at high performance solutions. From there it can gradually approximate the optimal energy efficient implementation.

A possible extension to the scheduler can be to traverse a Pareto curve instead. As discussed in subsection 2.3.1, applying such a method to configuration generation would pose the problem of finding the extreme points of the Pareto curve. However, this problem does not apply to schedule generation. For instance, the Pareto-tradeoff between performance and energy consumption could be explored using the scheduler[19].

4. Generation by schedule-based mapping

The proposed implementation generation method generates configurations and PASM code based on inputted schedules. The generation process maps instructions, produced values, and dependencies onto resources, and allocates additional resources if insufficient are available. Clustering instructions based on dependency requirements enables multiple instructions to be mapped simultaneously.

This chapter first discusses the goals of the implementation generator in section 4.1. Following that, the requirements for mapping individual instructions and live values are discussed in section 4.2 and 4.3 respectively. Dependencies are categorized by their requirements in section 4.4. The instruction clusters formed by these dependency requirements, referred to as instruction groupings, are discussed in section 4.5. Section 4.6 discusses impossible schedules. Section 4.7 discusses the resources that the schedule is mapped to. The chapter is concluded by sections 4.8, and 4.9, which respectively discuss the mapping process and the mapping requirements per mapping step.

4.1 Goals

The primary purpose of the proposed implementation generation method is to generate configurations and PASM for the CGRA based on inputted schedules. However, there are some requirements on its generating process and produced results. These are translated into goals and presented in this section.

The implementation generator is intended to be part of the overarching tool flow discussed in section 3.1. This is reflected in its goals. As the proposed tool flow heavily depends on the implementation generator, achieving these goals is essential for any further development.

4.1.1 Support schedule as-is

The schedule needs to execute exactly as provided.

The proposed searching method relies on the fact that the schedule is supported and executed exactly as-is, pertaining to performance and functional correctness. This provides accurate feedback to the scheduler regarding the effect of variations in the schedules, which benefits predictability of future variations.

4.1.2 Minimize number of FUs

The generated configuration needs to contain as few FUs as possible.

The proposed searching method expects the implementation generator to generate the most energy efficient implementation for its proposed schedule. As discussed in subsection 3.2.3, in the context of supporting a fixed schedule, this is achieved through minimizing the number of resources for each resource type, such as FUs.

4.1.3 Minimize number of connections

The generated configuration needs to contain as few connections as possible.

Similar as for FUs, minimizing the number of connections allows for the reduction of energy consumption.

4.1.4 Short computation time

The implementation generation tool needs to produce a result quickly.

This goal is rather subjective, and has been established through estimations. The implementation generation tool is used repeatedly in the proposed tool flow. The proposed tool flow iterates through many different schedules, each of which passes through the implementation generator to gain feedback. This feedback is used to steer the tool flow in the right direction as much as possible, and is essential to the searching process. As the implementation generator is used repeatedly, its computation time can contribute heavily to the total computation time of the searching method.

The strength of the proposed tool flow, compared to human development, is that it is capable of quickly exploring many possible implementations. Minimizing the computation time of the implementation generator is beneficial to this strength.

However, 'how fast' it should be is a debatable topic. The time spent computing a result depends on the work load that is provided, and how that work load is handled. Ultimately, the design space exploration of the searching method must outweigh the intelligence of a human developer. Human development of a configuration and PASM code can take days. However, the searching method might need numerous iterations. We have arbitrarily chosen that the implementation generator must be able to generate 100 configurations for an application that would cost a work day (i.e. 8 hours) for a human developer. This leads to a little less than 5 minutes per attempt. The benchmarks used for testing contain applications that could take more than a day to develop. Therefore, the goal for this thesis will be that each test case is generated within 5 minutes.

4.2 Instructions

Instructions need to be mapped to FUs of appropriate type. There are some instructions that can be mapped to multiple FU types (e.g. the 'pass' instruction can execute on LSUs, MULs, and ALUs), but many are limited to only one.

Instructions are distinguished by opcode and behavior, as identical opcodes do not always guarantee identical behavior (e.g. the 'lrm' instruction can execute on both the RF and LRM, but behaves slightly differently).

During the mapping process, instructions can have three self explanatory states, which will become relevant in the section on dependencies.

- Unmapped
- Mapped
- Being mapped

Mapping an instruction requires that its produced values and dependencies are mapped as well. Dependencies have different requirements depending on the mapping state of both parties (i.e. the producing and consuming instruction). Value mapping requirements and dependency mapping requirements are discussed in sections 4.3, and 4.4 respectively.

4.3 Values

This section will discuss values, live values, what their life span is, and what resources they are mapped to. The life span of values largely depends on related dependencies; however, will be discussed in this section.

4.3.1 Values

When an instruction is executed, it can produce one or more values. For instance, 'add outX, inY, inZ' writes a value to 'outX'. And, 'lgi BYTE, outX' writes a value to an IR (the implicit loading address), and one to 'outX'. These values are often used by other instructions. A dependency between instructions indicates that an instruction requires one of the values produced by another instruction.

It is possible for an instruction to depend on the values from multiple different producing instructions **for a single operand** (e.g. the value available on 'inY' of 'pass outX, inY' could have been produced by several instructions). These values are generally available from the same source (e.g. an output port, or internal register). These producing instructions (collectively referred to as producer sets) are allowed to overwrite each other without violating dependencies. Which value is made available to the operand depends on which producing instruction (or forwarding instruction, which will be discussed in subsection 4.4.3) was executed last.

Note! 'operand' does not necessarily refer to an explicitly indicated input port or register, but also to implicit ones.

4.3.2 Live values

Live values are values for which it is still **possible** that they are read by an intended consuming instruction. While they are live, they should not be overwritten. This will not be allowed in the proposed generating method. Producing instructions that are allowed to overwrite each other shorten the life span, as being overwritten makes being read impossible. Moved values will be considered different values with different dependencies. When a value is dead, it has no claim on whatever resource it was holding and can be overwritten.

4.3.3 Life span

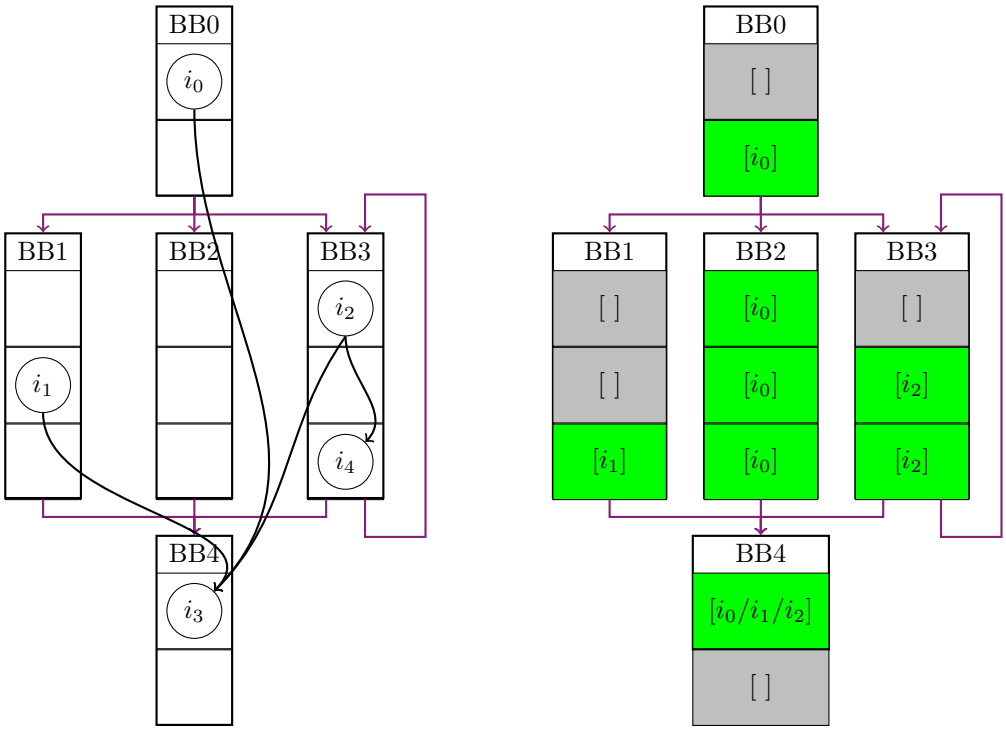
The life span of a value indicates when it is alive. Values are often produced as live values, but this is not necessarily always true. This does not necessarily mean the producing instruction is useless.

When a value is live depends on **possible** paths of the control flow. These possible paths can often be determined at compile-time, and therefore the life span of a value as well. Values are live on all timeslots of all possible paths from all possible production timeslots (these can be multiple due to branching) to all timeslots during which consumers execute, without being overwritten by allowed (other) producers. Live values can remain live after a consumer has been executed, as long as there is still a path possible that leads to a consumer.

Figure 4.1a shows an example schedule with some instructions and dependencies. Three instructions (i_0, i_1, and i_2) produce for the same operand of a shared consumer (i_3). Figure 4.1b shows when the values are live for each possible control flow path in that schedule (green), and which producing instruction has claim to the resource it was stored in.

As can be seen in the first timeslot of "BB4", multiple instructions can have claim to a shared resource. This is allowed and even necessary those instructions produce for the same operand of a shared consumer. Each value can be intended for other targets as well and still share the resource (as demonstrated by the dependency between i_2 and i_4). This will only result in dependency issues for impossible schedules, as discussed in section 4.6.

Values are written to internal registers, output registers, local memory, or global memory. These registers or memory slots need to guarantee live values are not overwritten, and need to conform to



(a) Schedule. Data flows: black arrows

(b) Live values

Figure 4.1: Value life time example

the requirements of dependencies on those values. More on registers and memory will be discussed in subsections 4.7.3 and 4.7.2 respectively, and section 4.4 explains more about dependencies.

Values can be written to unbuffered output ports as well. In this case, the value only lives for one cycle.

4.4 Dependencies

Dependencies require specific resources in the configuration to be available. However, not each dependency has the same requirements.

Categorized by their requirements, the dependencies can be grouped into four different classes for the Blocks CGRA: output register dependencies, internal register dependencies, forwarded internal register dependencies, and local memory dependencies. Each of these dependencies are discussed subsections 4.4.1, 4.4.2, 4.4.3, and 4.4.4 respectively.

Applications can also contain global memory dependencies. These deal with writing and reading values stored in global memory. However, there are no resources to reserve for these dependencies. As global memory is a globally shared resource, and each LSU can reach all its addresses. The application (and possibly the input data) specifies where data is written to and read from. The implementation generator has no control over this, and any dependency is therefore considered an emergent property. When and where data is written to or read from in global memory fully depends on the application, and not on the mapping. The only resource that needs to be reserved is an LSU to execute the instructions, but this is discussed in the "Instructions" section (4.2). This dependency type has no consequences for the implementation generator and will not be discussed

further.

4.4.1 Output register dependencies

Instructions can write a value to an output register (OR). These instructions will be referred to as OR producers. Instructions that depend on values stored in ORs will be referred to as OR consumers. These are the two parties of an OR dependency relation.

Figure 4.2 shows an example of how an OR dependency can be mapped, and shows the resulting data flow. As can be seen, the dependency revolves around the OR (red square). OR producers write their value to it, and OR consumers read from it. This requires a connection (red arrow) from the related output port to one of the input ports of the FU that executes the OR consumer. As discussed in section 4.3, the OR must retain the live value while it could be depended upon.

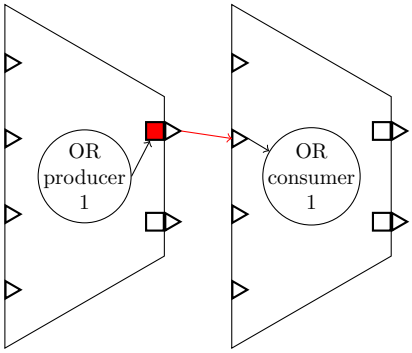


Figure 4.2: Example of OR dependency

The unbuffered ALU is an exceptional case. It has an unbuffered output port, which means there is no OR connected to it. Due to the similarities, dependencies regarding this unbuffered output port will also be considered as OR dependencies. The only difference is that the value life span will be one cycle.

OR dependencies are specified per OR producer, OR consumer and operand. Multiple OR consumers can depend on the same OR producer, but a single OR consumer can also depend on the same OR producer with multiple operands as illustrated by figure 4.3.

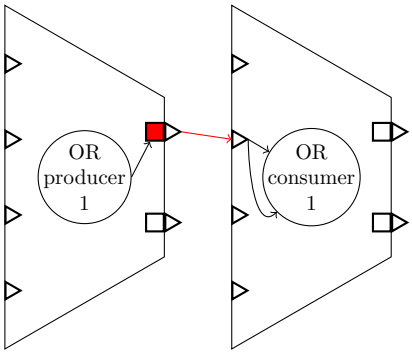
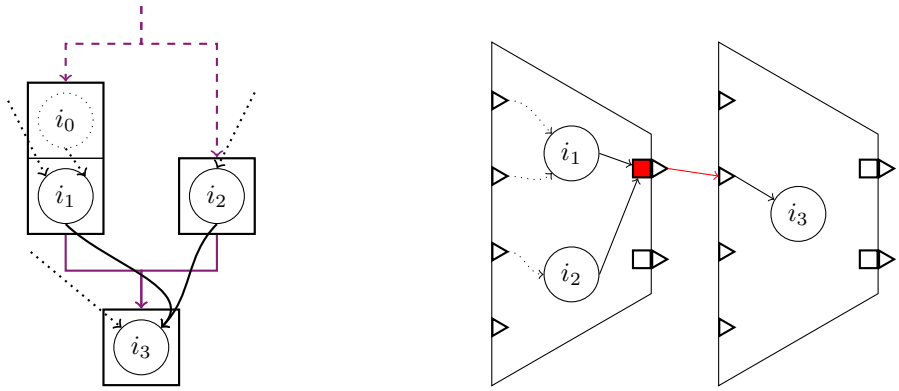


Figure 4.3: Multiple OR dependencies between identical OR consumer and OR producer

Additionally, a single operand of a single OR consumer can depend on multiple OR producers.

Which value will be used depends on which OR producer was executed last. These producers together are referred to as OR producer set. Figures 4.4a and 4.4b show an example of a schedule containing such an OR producer set (i_1 and i_2), and how this relates to the mapping. OR producer sets are required to write to a shared OR to prevent dependency violations.



(a) CDFG with OR producer set (i_1 and i_2); purple arrows: control flow; black arrows: OR data flow

(b) OR producer set mapping

Figure 4.4: Example of an OR producer set

4.4.2 Internal register dependencies

Internal register (IR) dependencies are quite similar to OR dependencies, except for the required connection. Some parts of this explanation will therefore sound like a rerun of the OR dependencies.

Some instructions write to one or more IRs (e.g. the 'srm rX, inY' instruction of the LSU stores the value provided for operand 'inY' in IR 'rX'). These are referred to as IR producers. Instructions that depend on values stored in IRs are referred to as IR consumers. An IR dependency consists of an IR producer, the IR to which it writes, and an IR consumer that reads from it. For each unique combination, a unique IR dependency exists (IR producers and IR consumers can write and read from multiple IRs, and could have multiple IR dependencies between them). Some instructions implicitly write to and read from IRs (e.g. instructions using implicit addressing in LSUs). These instructions are also IR producers and consumers.

Figure 4.5 shows an example of how an IR dependency is mapped, illustrated as a data flow. An IR producer writes to an IR (blue square), and an IR consumer reads from it. IRs are depicted by the row of squares with solid edges. IR dependencies require both the IR producer and IR consumer to be mapped to the same FU. Additionally, the IR which is written to and read from must be able to retain the live value without it being overwritten.

Figure 4.6a shows an example of an IR producer set. IR producer sets contain IR producers (i_1 and i_2) which produce to the same IR, and for the same IR consumer (i.e. i_3). As the figure illustrates, which value will be used depends on the control flow. Figure 4.6b shows how the example IR producer set needs to be mapped. Like for any IR dependency, all parties need to map to the same FU. This has a recursive effect.

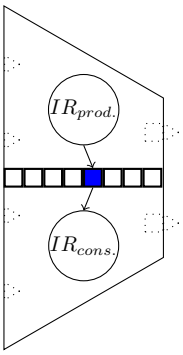
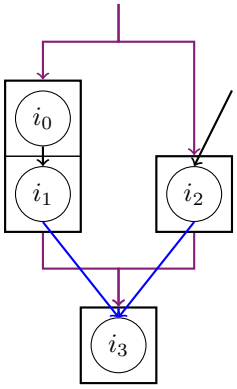
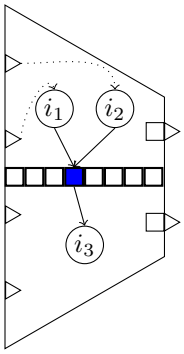


Figure 4.5: Example of IR dependency



(a) CDFG with IR producer set (i_1 and i_2); purple arrows: control flow; blue arrows: IR data flow; black arrows: OR data flow



(b) IR producer set mapping

Figure 4.6: Example of an IR producer set

4.4.3 Forwarded internal register dependencies

Forwarded internal register (FIR) dependencies are a rather unique case, which are related to the register file. The register file in the Blocks CGRA has the capability of **forwarding** the value in an IR through an output port, rather than copying its value to an OR. It has no ORs attached to its output ports.

Figure 4.7 shows a crude schematic overview of how the IRs in a register file relate to its output ports. As can be seen, output port 0 and 1 are affected differently.
Output port 1 can forward different IRs. Which IR is forwarded, is controlled using the 'lrm' instruction. Changing this forwarding takes no cycles whatsoever and can execute simultaneously with instructions that depend on its forwarded value.
Output port 0 always forwards IR 0. Storing data in IR 0 makes it available on output port 0, essentially making it serve as an OR. However, IR 0 can also be forwarded by output port 1, so special care still needs to be taken.

There are three parties in a FIR dependency.

- The FIR producer, which stores data in an IR.
- The FIR forwarder, which forwards an IR.
- The FIR consumer, which is the instruction that reads the forwarded data.

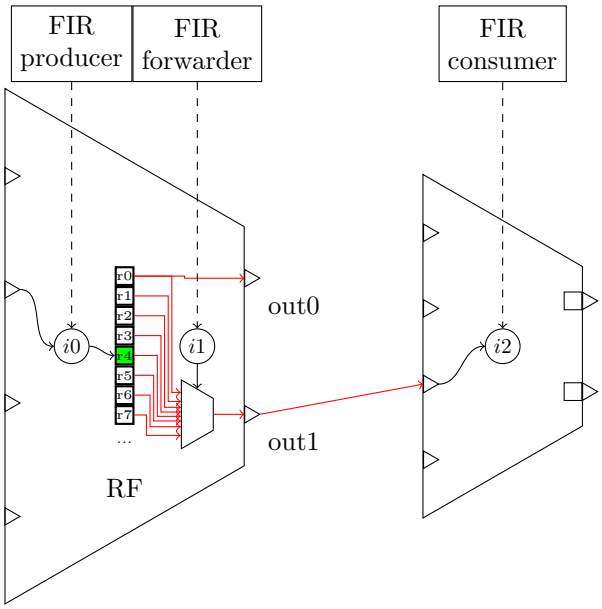


Figure 4.7: Example of FIR dependency; red: physical connection

Figure 4.7 illustrates the three parties of a FIR dependency. Instruction i_0 is the FIR producer and stores data (from an input port) in an IR. Instruction i_1 is the FIR forwarder and controls the multiplexer that forwards an IR. Instruction i_2 is the FIR consumer, and consumes whichever forwarded value arrives on its input port.

FIR dependencies are created for each unique combination of FIR producer, FIR consumer, FIR consumer operand, and IR index. A FIR dependency can have multiple FIR forwarders. The FIR producer and FIR forwarders need to map to the same FU.

The relation between FIR forwarder and FIR consumer is similar to that of OR producer and OR consumer, respectively. The FIR forwarder sets data (i.e. the forwarding address) which is depended upon by the FIR consumer. It is not read, but the value still needs to be correct when the FIR consumer executes. Therefore, the life span of the forwarding address is the same as it would have been a value for an OR consumer. Additionally, a connection is required. This similarity is abused by defining an OR dependency between the FIR forwarder and FIR consumer, which reserves the forwarding address register for the appropriate life span, and allocates a connection.

The relation between FIR **producer** and FIR consumer is also similar to that of OR producer and OR consumer, respectively. However, there is a subtle difference. The FIR producer writes a value (i.e. the value stored in the IR) that remains alive until there is no possibility left that an intended FIR consumer will **use it**. It is used when the FIR consumer executes while the value is still stored in an IR **and** that IR is being forwarded.

Like for IR and OR dependencies, there exist FIR producers that produce on the same IR, for the same operand of a shared FIR consumer. These producers make up a FIR producer set and are allowed to overwrite each other.

Subcase: output port 0 dependencies

The FIR dependency also applies to FIR consumers that depend on output port 0. Figure 4.8 shows an example for this subcase of the FIR dependency. What can be seen is that no FIR forwarders exist for this relation. This is because output port 0 always forwards IR 'r0', and no

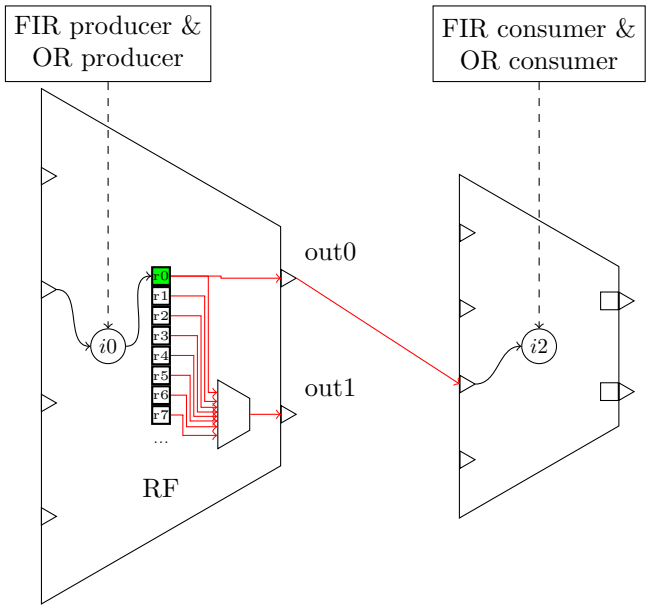


Figure 4.8: Example of FIR dependency, implemented as OR dependency; red: physical connection

FIR forwarder is needed.

The instructions that write to 'r0' are classified as OR producers as well as FIR producers. This is because their produced values are automatically made available on output port 0. By classifying the FIR consumer as an OR consumer, an OR dependency can be established. The life span of the value produced by the FIR producer is identical if it were an OR producer, and the dependency requires a connection. This OR dependency will deal with setting up the connection.

However, the OR dependency reserves a non-existent OR, instead of the IR. This was fine for the FIR forwarders on output port 1, as it refers to the forwarding address. However, instructions that write to IR 0 need to know that specific IR is reserved. Therefore, the FIR dependency still exists to reserve IR 0 for the life span of the value.

4.4.4 Local memory dependencies

Local memory (LM) dependencies are similar to IR dependencies and largely have the same requirements. However, the main difference is that LM dependencies deal with a range the local memory, rather than individual registers.

LM producers are instructions that write to local memory. LM consumers are instructions that read from local memory. An LM dependency exists between LM producer and LM consumer if there is any chance the LM consumer reads from the same memory addresses as the LM producer has written to. For the Blocks CGRA, these dependencies are specific to LSUs, which are the only FUs with access to local memory. LSUs can not access local memory of other LSUs. Therefore, both participants of an LM dependency must be mapped to the same LSU.

The complicating factor for LM dependencies are the implicit storing and loading instructions. The addresses used by these type of instructions are not necessarily known to the implementation generation tool, as they are subject to change and could be influenced by application input data. Therefore, which memory addresses need to be reserved can not be guaranteed at compile-time.

Instead, LM dependencies indicate which instructions need to map to the same LSU, and which LM dependencies should not map to the same LSU. This satisfies LM dependencies, while providing the ability to prevent live value overwrites.

4.5 Instruction groupings

The dependencies pose specific requirements on the mapping of instructions. For instance, some instructions will need access to the same IR as others, or write to a shared OR. These requirements resulted in four distinct groupings of instructions, one for each type of dependency. These groupings are an emergent property of the dependency requirements, and indicate which resource type needs to be shared by the instructions in these groupings.

The groupings are: output register linked groupings, internal register linked groupings, forwarded internal register linked groupings and local memory groupings. These are discussed in subsections 4.5.1, 4.5.2, 4.5.3, and 4.5.4 respectively.

The grouping types have one requirement in common: all instructions in a grouping need to map to the same FU. Because of this, overlapping groupings (of any type) need to map to the same FU. This leads to the introduction of the final type of grouping: the FU linked grouping. This grouping is discussed in subsection 4.5.5.

Each instruction is part of each of the five grouping types. This is regardless of it having any related dependencies (e.g. an instruction that does not access any IR is still part of an IR linked grouping with a size of one).

4.5.1 Output register linked groupings

The OR linked groupings consist of instructions that are linked to a shared OR by OR dependencies. If the OR linked grouping contains OR producers, then all of them **need** to write to the same OR. If any two (or more) instructions in an OR linked grouping write to separate ORs, then one or more OR dependencies will inevitably be violated.

OR linked groupings are established based on OR producer sets. All instructions in an OR producer set belong to the same OR linked grouping. Overlapping OR producer sets also belong to the same OR linked grouping. Figure 4.9 shows an example of this. The orange and blue rectangles enclose two OR producer sets, and instruction i_1 is part of both sets. OR linked groupings are the collective result of all overlapping OR linked producers.

The remaining instructions (i.e. the instructions that are not OR producers, and therefore can not be part of any OR producer set; i_5 and i_6 in the figure) are also part of OR linked groupings. They will be the only instruction in their respective groupings. This might seem counter-intuitive; however, the groupings consist of instructions that are **linked** by OR dependencies. This does not require them to actually write to ORs or be part of any OR dependency.

When mapping any instruction in an OR linked grouping, all of its instructions must map to the same FU and write to the same OR linked grouping (if it contains OR producers). This means that all requirements regarding each of the instructions, their dependencies, and live values must also be satisfied by the FU, and OR if it contains OR producers. This enables the mapping process to simultaneously map all instructions in a single OR linked grouping.

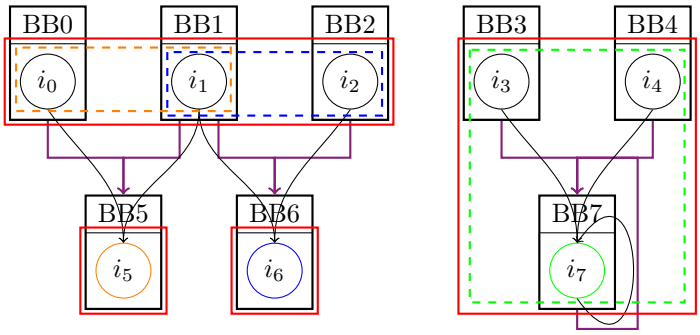


Figure 4.9: OR linked groupings: solid red. OR Producer sets: dashed. Data flows: black arrows

4.5.2 Internal register linked groupings

All parties of an IR dependency (i.e. the IR producer and IR consumer) need to have access to the same IR. This is only possible if they are executed on the same FU. This has a recursive effect. As illustrated by figure 4.10, if each IR dependency relation forms an edge in a mesh, and the participating instructions (i.e. the IR producer and IR consumer) form the nodes, then all nodes in the mesh need to be mapped to the same FU. These are the instructions that make up IR linked groupings. The IR linked groupings are shown as blue rectangles.

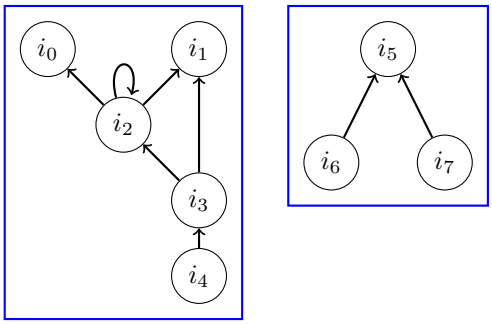


Figure 4.10: Mesh of IR dependencies; IR linked groupings in blue

If one of the instructions in an IR linked grouping is mapped, then all instructions in the grouping need to map to the same FU. Therefore, the requirements of all instructions, dependencies, and values of the IR linked grouping need to be satisfied by the FU as well. Some OR and FIR dependencies require a mapping to an OR or output port. This is not necessarily determined for all instructions within an IR linked grouping when one of its instructions is mapped. Therefore, the requirements of these dependencies can not necessarily be checked until their related instructions are mapped themselves.

4.5.3 Forwarded internal register linked groupings

FIR linked groupings are the groupings of instructions that need to map to the same FU due to FIR dependencies. FIR dependencies require some instructions to have access to a shared IR, and/or a shared output port. The FIR producer writes a value to an IR, and a FIR forwarder needs access to the output port that forwards it. Therefore, both need to be mapped to the same FU.

Like for the IR dependencies, this has a recursive effect. Any overlap of FIR producers and/or FIR forwarders between FIR dependencies means all producers and forwarders of the overlapping dependencies belong to the same FIR linked grouping. This includes the case where a FIR producer is also a FIR forwarder for another dependency (e.g. the 'lrm_srm rX, rY, inZ' forwards 'rX', while writing the value on 'inZ' to 'rY').

Additionally, the FIR dependencies indicate which operand of which FIR consumer the dependency pertains to. If multiple FIR producers produce for the same operand of a shared FIR consumer (note: They do not necessarily have to produce to a shared IR), then their values must be forwarded through the same output port. The operand of the consumer will always receive its value from the same output port. These FIR producers (and their related FIR forwarders) therefore also belong to the same FIR linked grouping.

The instructions in FIR linked groupings are often not mapped simultaneously. However, if one of the instructions is mapped, then all instructions in the grouping need to map to the same FU. Therefore, the FU must satisfy the requirements of all instructions, dependencies, and values in the same IR linked grouping. Some OR and FIR dependencies require an OR or output port to be selected. However, mapping one instruction in the FIR linked grouping does not necessarily determine that for all instructions in the FIR linked grouping. Therefore, the requirements of these dependencies can not necessarily be checked until their related instructions are mapped.

4.5.4 Local memory linked groupings

LM linked groupings consist of all instructions that need access to the same local memory, and are therefore all mapped to the same load store unit.

Both the LM producer and LM consumer of an LM dependency must have access to the same local memory. This has a recursive effect. Some LM consumers will have have multiple LM producers from which they consume, some LM producers have multiple LM consumers consuming their data. Some operations (e.g. the 'lli_sli' instruction) read and write to local memory simultaneously, and can be LM producer and LM consumer simultaneously. All instructions (indirectly) related to each other through LM dependencies belong to the same LM linked grouping.

The instructions in LM linked groupings are often not mapped simultaneously. However, if one of its instructions is mapped, then all instructions in the grouping need to map to the same FU. Therefore, if an instruction of a LM linked grouping is mapped to a FU, it needs to satisfy the requirements of all instructions in the grouping, as well as the requirements of their dependencies and values. Again, an exception pertains to OR and FIR dependencies, as these require an OR or output port to be specified. This is not necessarily determined for all instructions in LM linked grouping when one (or more) of its instructions are mapped. Therefore, their requirements can will only be checkable when their related instructions are mapped.

4.5.5 FU linked groupings

Each of the previously discussed groupings (i.e. subsections 4.5.1, 4.5.2, 4.5.3, and 4.5.4) have one requirement in common: all of their instructions need to map to the same FU. If any groupings overlap, regardless of type, they will need to map to the same FU.

FU linked groupings consist of all groupings that need to map to the same FU. Each grouping, regardless of type, is always part of exactly one FU linked grouping. It is an emergent property of all grouping types discussed so far.

If any instruction in a FU linked grouping is mapped, then all instructions need to map to the same FU. The FU therefore needs to satisfy the requirements of all instructions in the FU linked

		Groupings				
Benchmark	Instructions	FU linked	OR linked	IR linked	FIR linked	LM linked
binarization/scalar	33	20	30	22	33	33
binarization/scalar_dynamic	33	20	30	22	33	33
binarization/tapeout1_dynamic	33	20	30	22	33	33
binarization/tapeout2_dynamic	33	20	30	22	33	33
binarization/vector4	130	73	121	78	130	130
binarization/vector8	278	165	261	174	278	278
binarization/vector8_dynamic	278	165	261	174	278	278
fir/parallel	267	227	242	251	267	267
fir/parallel_dynamic	267	227	242	251	267	267
iir/parallel	106	86	91	101	106	106
iir/parallel_dynamic	106	86	91	101	106	106
projection/vector8	395	223	365	273	383	339
projection/vector8_dynamic	395	223	365	273	383	339
SIMD/binarization	288	130	288	240	178	288
SIMD/fir_shift	419	293	403	406	321	419
tests/tapeout1_count10	27	18	25	20	27	27
tests/tapeout2_count10	27	18	25	20	27	27
VLIW/binarization	371	172	354	291	265	371
VLIW/fir	194	125	191	189	133	194
VLIW/iir	178	111	176	172	119	178

Table 4.1: Groupings per benchmark

grouping, and the requirements of their dependencies and values. Again, some OR and FIR dependencies are the exception. If an OR linked grouping is mapped, only then are the OR and/or output port determined for its instructions. FU linked groupings can consist of multiple OR linked groupings, so mapping one of them does not necessarily determine the OR or output port used for the entire FU linked grouping. Therefore, it is not necessarily known to which OR or output port the requirements of the OR and FIR dependencies apply, until the FIR or OR linked grouping of their producers and/or forwarders is mapped.

To sketch how many of each grouping type can be in an application, table 4.1 shows some numbers of the benchmarks used for the experiments (which will be discussed in chapter 6). Each instruction is part of exactly one grouping of each grouping type. The mapping process will map a single OR linked grouping per mapping step. However, as can be seen from the table, OR linked groupings with more than one instruction are rare. They will dictate the FU to map to for other OR linked groupings in the same FU linked grouping, limiting their choice in ORs during their respective mapping step.

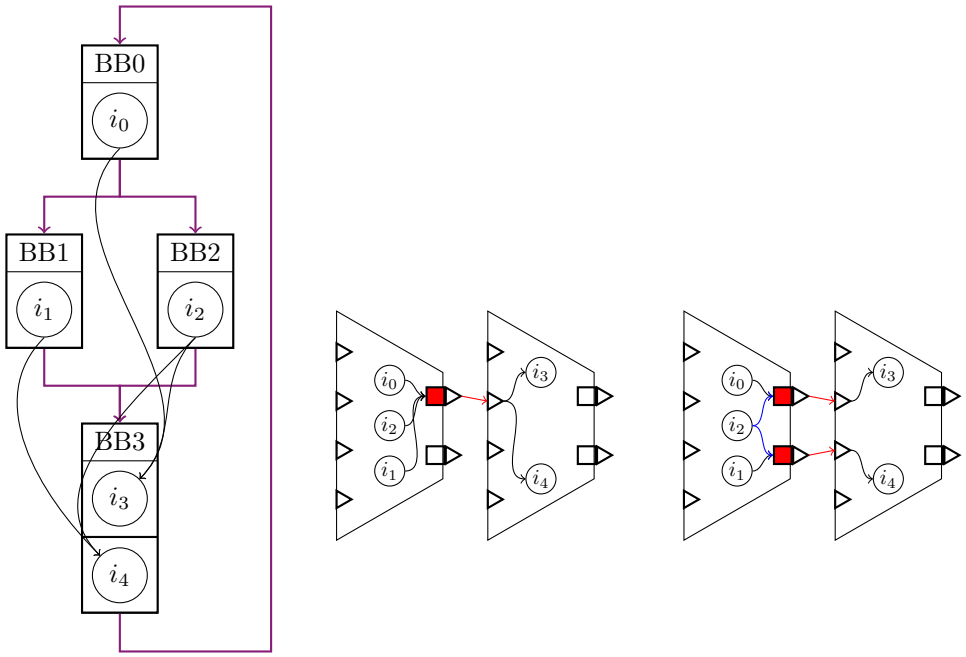
In the future it might be advisable to try simultaneous mapping of entire FU linked groupings. However, this would mean checking all combinations of (producing) OR linked groupings and ORs for each suitable FU per mapping step. This would result in $F * 2^R$ complexity per step, where F is the number of compatible FUs, and R is the number of OR linked groupings in the FU linked grouping. Currently, the complexity is F per mapping step.

4.6 Impossible schedules

Some schedules are impossible to satisfy using the CGRA without violating at least one dependency. These are the impossible schedules. A simple example is a FU linked grouping that contains two instructions that need to execute simultaneously. Simultaneously execution of two instructions on one FU is not possible for the Blocks CGRA. As such, the FU grouping can not be executed on a single FU, inevitably violating a dependency or failing the mapping.

Figure 4.11a shows another example of such an impossible schedule. Overlapping arrival points of the data flow arrows indicate these flows are provided to the same operand. Instructions $\{i_0, i_2\}$ make up a single OR producer set, and $\{i_1, i_2\}$ make up another. As they have overlap, they belong to the same OR linked grouping and need to be mapped to the same OR. However, i_0 is not allowed to overwrite the value of i_1 (and vice versa). Whatever mapping is tried, a dependency will be violated. Figure 4.11b shows how the OR linked grouping is satisfied, but then the possibility exists that i_0 and i_1 overwrite each other. The mapping as shown in figure 4.11c prevents this, but requires the i_2 instruction to write to multiple ORs, which is not possible for the Blocks CGRA at the time of writing.

An assumption is made that all OR producers in an OR linked grouping are allowed to overwrite each other. This simplified the mapping process, and only affects impossible schedules. An implementation would be generated with undefined behavior for such impossible schedules, instead of failing to map. These impossible schedules can be detected and refused before mapping.



(a) Impossible schedule. Data flows: black arrows

(b) OR producer mapping 1

(c) OR producer mapping 2 (blue: impossible data flow)

Figure 4.11: Impossible schedule example

4.7 Resources

The mapping process maps instructions, dependencies, and values to resources. This section will discuss those resources.

4.7.1 FU execution timeslot

Each FU in the Blocks CGRA can start execution of only one instruction per timeslot. FU execution slots can therefore be reserved for any possible timeslot in the schedule. This allows only one instruction to be executed on any FU at any timeslot. Some instructions take more than one cycle to finish execution. However, their execution process is pipelined and does not affect the availability of the FU in subsequent timeslots.

These timeslots are checked and reserved for each instruction in a FU linked grouping. Checking and reserving only needs to be done when mapping the first OR linked grouping of a FU linked grouping.

4.7.2 Memory

Two types of memory exist in the Blocks CGRA. These are global and local memory.

Global memory is not reserved by the implementation generator. It is a single globally accessible resource, and the accessed addresses are determined by the application and/or application input data. The mapping process of the implementation generator has no influence on which addresses of the global memory are accessed, nor does it gain anything by reserving them.

Local memory is also not reserved by the implementation generator. The inputted schedule indicates which LM dependencies are mutually exclusive (i.e. if mapped to the same local memory, they could cause a dependency violation). The mapping respects these indications, by mapping the LM linked groupings containing mutually exclusive LM dependencies to different LSUs.

4.7.3 Registers

Registers, regardless of type, can be reserved for each timeslot in the schedule. Timeslots are reserved to indicate the register hold a value, and/or to indicate that it is being written to.

It is possible for a value to be 'stillborn'. This means that, at the timeslot of production, there is no possibility that a consumer will ever read it. It is allowed to be overwritten by any other instruction. However, the production of this value does overwrite the contents of a register. Therefore, to guarantee that it can be overwritten, but will not overwrite others, the register will be in a different reservation state.

Registers can be in one of four reservation states per timeslot. Table 4.2 represents the states, and shows the conditions for each state. The register can be written-to, which means it overwrites the value that was present. The register can also be retaining a live value, which means what is in there is not allowed to be overwritten.

		Write moment	
		No	Yes
Contains live value	No	S1	S2
	Yes	S3	S4

Table 4.2: Register reservation states per timeslot

The following paragraphs explain a bit about the states and their requirements. These requirements assert that dependencies are respected, and the result of the mapping will conform with the intended functionality of the schedule.

S1

The first state is the default state. The register contains no live value, nor is it written to in that timeslot. The register could be used to retain a live-value (without it being overwritten), and/or to be written to (without it overwriting a live value).

S2

In this state, the register is reserved by an instruction that produces a 'stillborn' value. The produced value will not be used. The register is not able to retain a live value during this timeslot, as it would be overwritten. However, it is possible to write another stillborn value to it during this timeslot. This is possible when two basic blocks branch into one.

S3

This is the state in which a live value is retained in the register (which was written before the current timeslot). It is not allowed to be overwritten, nor is it capable of storing another live value. Producers from the same producer set are allowed to also reserve the register, if it was in this state. Figure 4.1 in subsection 4.3.3, regarding the life span of live values, illustrates the occurrence of such a case in timeslot 1 of "BB4".

S4

This is the state in which a live value is written into the register. It is not allowed to be overwritten, nor is it capable of storing another live value. Producers from the same producer set are allowed to also reserve the register in this state.

Possible improvement

The above states and their requirements aim to assert that dependencies are respected. However, there are cases where the requirements are 'overzealous', and the registers could be used by other unrelated instructions without problems.

Figure 4.12a illustrates an example where the requirements are overzealous. Both instructions i_0 and i_1 write to an OR. Both instructions produce their value in the cycle after its execution.

Figure 4.12b shows the reservation state per timeslot of an OR, after mapping the value produced by i_0 to it. The value is stillborn in "BB3", because no control flow path exists from "BB3" to i_2, which is the only consumer for i_0. Despite that, the register is put in state S2, as it is overwritten. Instruction i_1 is not allowed to reserve the same OR, because of the reservation state of the register in "BB3". However, the value produced by i_1 could never be overwritten by i_0. Either "BB0" or "BB1" executes, and no control flow path exists from "BB3" to "BB0". It would therefore be safe to use the same OR.

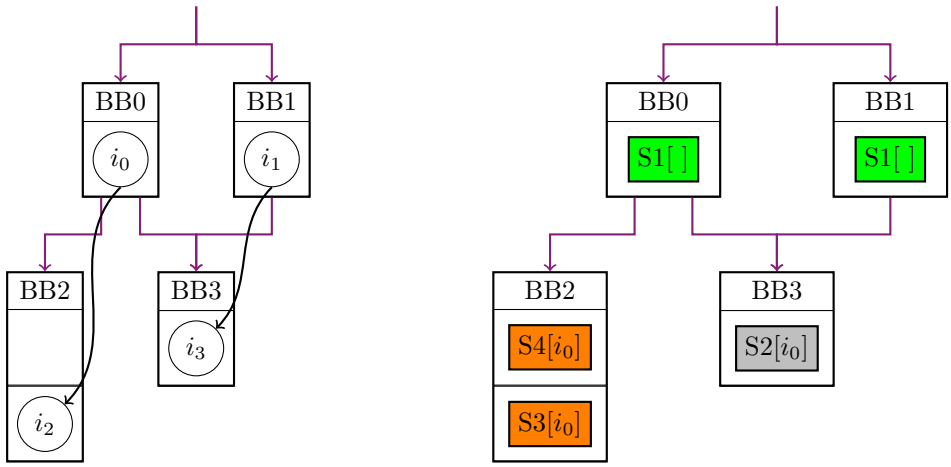
The discussed case is a rare occurrence. The example case allows for i_3 to execute using an undefined input, which happens when "BB0" is executed before "BB3". This leads to undefined behavior, which generally is undesirable in applications.

4.7.4 Input ports

When mapping an instruction, both its incoming and outgoing dependencies need to be satisfied. If that instruction is an OR or FIR consumer, input ports need to be used to satisfy their dependencies. Each FU in the Blocks CGRA has at most four input ports, and each input port can have only one incoming connection.

Input ports have three states.

- Free.
- Reserved.
- Connected to.



(a) CDFG (b) States after mapping i_0

Figure 4.12: Exception case

The input port is free, if it is not connected to, nor reserved. This is the default state of each input port.
An input port is connected to, if an incoming connection is connected to it.
An input port is reserved, if the consumer is mapped to its FU and it is yet unknown where the producer will write its value to. This means that it is not yet known to which output port the input port will be connected. It might turn out that an existing connection will be used for this, in which case the reservation is canceled without utilizing the input port. Input ports are reserved for the entire FIR or OR linked grouping of which the producers are part of.

FU linked groupings can have multiple consumers that depend on multiple unique FIR or OR linked groupings. It is possible that these groupings are in greater number than the input ports on a FU. This requires 'overbooking' of the input ports (i.e. more are reserved than exist). If such an overbooking occurs, **all** input ports must be in the "Free" state prior to the reservation. Overbooking can only be resolved if multiple of the depended upon FIR and/or OR linked groupings write to the same output port. This reuses connections and therefore the total required input ports.

4.8 Mapping process

The mapping process consists of a number of phases. These are described in this section. First an initialization phase, followed by a mapping phase during which the configuration is generated, and finished with PASM generation.

4.8.1 Initialization phase

First, the different groupings are discovered based on the dependencies. The groupings are instrumental to the mapping process. Following that, a list of OR linked groupings is composed and ordered. Several ordering methods were tested, and are described in section 6.2. When this is done, the mapping can commence.

4.8.2 Mapping phase

The actual mapping phase consists of mapping steps. Each mapping step maps an OR linked grouping to a FU, and if it contains an OR producer, to an OR. A mapping step itself consists of

multiple phases as well, described in the paragraphs below.

Checking and scoring mapping options

The mapping step starts by collecting the various mapping options for the OR linked grouping. What constitutes as a mapping option depends whether the mapping OR linked grouping contains an OR producer or not. If it does, then mapping options pertain to combinations of FUs and their ORs (i.e. a FU to execute the instructions, and OR to store their produced values in). If it does not contain an OR producer, then FUs are considered mapping options.

FUs need to satisfy all the requirements of the overarching FU linked grouping, all its instructions, their dependencies, and produced values. If the OR linked grouping contains OR producers, then the OR needs to satisfy the requirements of the produced values. For specifics regarding these checks, see section 4.9.

Mapping options are scored on their 'suitability' for the OR linked grouping. This takes into account the dependencies of the overarching FU linked grouping. Two different scoring heuristics were tested, and are explained in subsection 6.4. Both aim to minimize the number of required connection allocations. Other CGRA mapping processes also factor connection usage into mapping costs [13], albeit in a different manner and for a different purpose. In our case, reducing connection allocation enables FUs to retain more available input ports that can be used for other dependencies. This increases the probability of the FUs to be viable mapping options for other mapping steps.

Mapping options do **not** pertain to combinations of FUs and their IRs, nor is it required. Currently, the implementation generator is not allowed to choose which IR is used to store values, as this is dictated by the dependencies in the schedule. However, this is not necessary and can be improved, but mapping options would still not include IR choice. To implement this, **specific** IRs would not be chosen during the mapping step itself. IRs can be reserved for (F)IR dependencies, but which specific one is used will be determined after the mapping phase is complete. During the mapping phase, the number of live values (in IRs) during any timeslot in a single FU, would be limited to the number of IRs in that FU. After mapping is complete, specific IRs are chosen using graph coloring. Because ORs are connected to input ports, this method does not apply to ORs.

Mapping selection and application

Based on the scores, the best mapping option is chosen. The OR linked grouping is mapped to it, and the required resources are reserved. If needed, connections are allocated. If no mapping options are available, a FU appropriate to the OR linked grouping is allocated, checked and scored. If this FU fails its checks as well, the problem is caused by a preceding mapping step, and backtracking commences. Otherwise, the next mapping step is started.

Backtracking

Backtracking reverts to the preceding mapping step. After reverting to the preceding mapping step, the 'Mapping selection and application' subphase is started. It will choose the next mapping option in its queue of scored mapping options. Using the new mapping, the current mapping step is retried. However, the preceding mapping step could also initiate backtracking if all its mapping options (including a newly allocated FU) have been tried.

The problem with backtracking is that it grows incredibly slow, very quickly. Per reverted mapping step, it will retry all subsequent mapping steps until it is no longer blocked. Equation 4.1 illustrates how the runtime grows as the backtracking depth d gets larger. i indicates a specific mapping step. S_i indicates a weighted average of mapping options per mapping step. The number

of mapping options can vary if preceding mapping steps are altered, therefore S_i is a weighted average, instead of the number of mapping options themselves. Finally, M is the complexity of a single mapping step.

$$M * \prod_{i=1}^{d} S_i \tag{4.1}$$

From the equation, we can see the runtime grows exponentially as the backtracking depth grows larger. During some initial tests of the implementation generator, we could see that a backtracking depth of about 23 mapping steps was already responsible for 3 days of computation work.

To prevent the implementation generator from spending days on a single mapping problem, an upper limit to the backtracking depth was implemented. If that limit is reached, the mapping phase is restarted with an altered order of OR linked groupings. Different methods of reordering were tested, and are discussed in section 6.3.

4.8.3 PASM generation phase

PASM is generated based on the mapping, the schedule, its instructions, and dependencies. All instructions that map to the same FU are added to the instruction memory of its instruction feed. Their timeslots are dictated by the schedule. The explicit operands of instructions are determined by the connections of the FU and the mapping of their consumers or producers. Operands indicating registers, as well as value types, are supplied by the schedule.

4.9 Mapping requirements

To be able to map an OR linked grouping to a FU and/or OR, a number of requirements will need to be satisfied by the FU and/or OR. These requirements are posed by the instructions, dependencies, and values of the overarching FU linked grouping.

If the OR linked grouping contains OR producers (and if it does, it **only** contains OR producers), then the FU **and** OR will be subjected to checks. If the OR linked grouping contains only non-OR-producing instructions (in which case it always has only 1 instruction), then **only the FU** will be subjected to checks. The subsections below discuss the requirements that are checked.

The availability of each resource is time dependent. For instance, the registers can be reserved to contain a live value for one timeslot, and can be available again in a later timeslot. This allows the resources to be modeled in a similar fashion as the graph-based resource model used by [18].

4.9.1 FU type

The FU must be of appropriate type to be able to execute all instructions in the overarching FU linked grouping.

4.9.2 FU execution slot availability

As explained in subsection 4.7.1, FUs can start execution of 1 instruction per timeslot, regardless of the execution time of preceding instructions. For each time slot, an execution slot is reservable. An execution slot must be available for each instruction in the overarching FU linked grouping.

4.9.3 FU linked instructions

If any instruction in the overarching FU linked grouping is already mapped, then the OR linked grouping that is being mapped must map to the same FU.

4.9.4 FU input port checks

Connections are used to satisfy OR and FIR dependencies. OR and FIR consumers must be able to read values from the output port to which the producers write. Therefore, the FU to which the consumer is mapped requires a connection from that output port to one of its input ports. However, those input ports are limited, and therefore, so are the incoming connections.

During the mapping process, six different cases regarding dependencies can be encountered. Different requirements apply for each case.

Figure 4.13 illustrates the six dependency cases that can be encountered when mapping generic example instruction i_0. As can be seen, the cases are distinguished by the mapping state of 'the other party' (i.e. the instruction with which i_0 has a dependency relation), and whether that other party is a consumer or producer for i_0.

The 'mapping state' of the other party is based on a different grouping depending on whether it is a producer or consumer. This is based on the required information to be able to determine whether the dependency can be satisfied.

The mapping state for producers is based on the mapping state of the OR linked grouping. If its OR linked grouping is mapped, then the output ports to which the producers write their values are known.

The mapping state for consumers is based on the mapping state of the FU linked grouping. FU linked groupings are 'mapped' if any of their OR linked groupings are mapped. If the FU linked grouping of the consumer is mapped, then the FU to which a connection must exist is known.

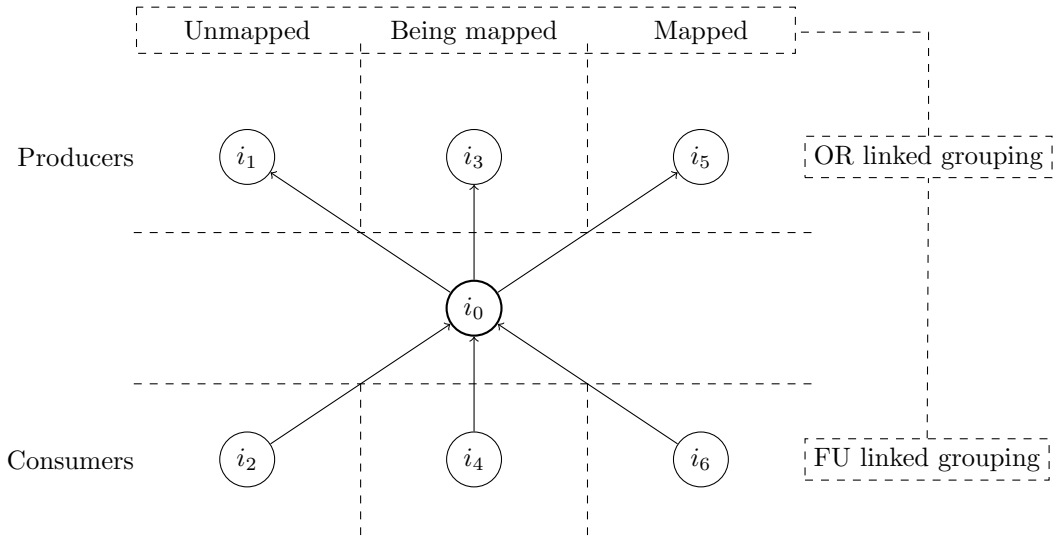


Figure 4.13: Dependency cases

The different cases are described in the following paragraphs. The paragraph titles indicate the status of the other party of the dependency. For instance, "Mapped producer" indicates that i_0 is a consumer, and that it depends on a producer that has already been mapped. The i_0 instruction is always considered as 'being mapped', as that is when the checks occur.

Unmapped producer

The case of the unmapped producer is illustrated by the dependency relation between instruction i_1 and i_0.

In this case, no information is available on which output port i_1 will write to. Therefore, an input port is reserved for the OR linked grouping that i_1 is part of. Only one input port is reserved per OR linked grouping. Reservations are made for all unmapped OR linked groupings that produce for instructions in the FU linked grouping that i_0 is part of. Input ports can be overbooked (i.e. more input ports are reserved than the FU has), but only if all input ports are free to connect to. If input ports fail to reserve, the FU fails the check.

When it is the turn for the producer to be mapped, this will present itself as the case "Mapped consumer" and is discussed in one of the following paragraphs.

A special instance of this case is when i_1 is part of the same FU linked grouping as i_0. However, for producers only the mapping state of the OR linked grouping matters, therefore this still falls within the scope of the "Unmapped producer" case. This means that the dependency will be mapped to a self connection. This changes nothing to the approach: an input port needs to be reserved for the OR linked grouping of the producer.

Being mapped producer

The producers that are being mapped are part of the same **OR linked grouping** as the instruction that is being mapped. This case is illustrated by the dependency relation between producer i_3 and consumer i_0. This also includes the case where instructions i_3 and i_0 are the same instruction (i.e. a self-dependency).

This dependency requires the considered FU to have a self connection from the OR that is considered, or to have input ports available for the self connection. If insufficient input ports are available and such connection does not already exist, the check fails.

Mapped producer

The case of the mapped producer is illustrated by the dependency relation between instruction i_5 and i_0. The OR linked group of i_5 has already been mapped. From that, we know which output port the depended upon data is written to. The FUs that are checked will need an input port available to connect to that output port.

An input port needs to be available for each unique output port to which the producers have been mapped. This holds for all OR producers for the FU linked grouping that i_0 is part of. If the FU does not have sufficient input ports available, the FU fails the check.

The producers can be mapped too wide spread (i.e. mapped to more output ports than the FUs have input ports). In this case, **all** FUs fail and the only solution is to do backtracking.

Unmapped consumer

The case of the unmapped consumer is illustrated by the dependency relation between instruction i_0 and i_2. At the time producer i_0 is being mapped, the FU linked grouping to which consumer i_2 belongs has not been mapped to a FU yet (i.e. no instruction in that FU linked grouping has been mapped yet).

This case has no direct requirements. The output port can connect to theoretically unlimited number of input ports. Whether i_2 can connect to this output port depends on which FU it

maps to. However, the producers can be too wide spread, which will inevitably lead to a blocked mapping process. This is best caught in this mapping step, but at the time of writing is not checked.

Being mapped consumer

The consumers that are being mapped are part of the same **FU linked grouping** as the instruction that is being mapped. This case is illustrated by the dependency relation between instruction i_0 and i_4. This also includes the case where i_4 and i_0 are the same instruction (i.e. a self-dependency).

This dependency requires the considered FU to have a self connection on the OR that is considered, or to have input ports available for the self connection. If insufficient input ports are available, the check fails.

This case is identical to "Being mapped producer", with the exception that i_4 does not necessarily has to be in the same OR linked grouping as i_0.

Mapped consumer

The case of the mapped consumer is illustrated by the dependency relation between instruction i_0 and i_6. At the time producer i_0 is being mapped, the FU linked grouping to which consumer i_6 belongs has already been mapped to a FU. The consuming FU will need an input port unconnected (disregarding reservations), or a connection needs to exist from the considered output port to the FU that executes i_6.

Note that no attention is given to the reservations here. The reservations of input ports on the FU executing i_6 are free to be ignored for this case. Reserving input ports only serves to keep them available for the reserving OR linked groupings. Additionally, it is impossible to arrive at this case without i_0 reserving an input port in the preceding "Unmapped producer" case. However, due to the possible overbooking of input ports, the reservations do not guarantee input ports. Therefore, at this point, the dependency requires an actual unconnected input port, rather than a reservation.

4.9.5 OR availability

OR availability is determined by whether an OR can retain the values produced by OR producers in the being-mapped OR linked grouping, without any live value being overwritten. If this can be guaranteed, the OR passes the check.

Section 4.3 explains more about the lifespans of live values, and subsection 4.7.3 explains more about when an OR is available, and when it can be written to. This is a check performed for OR linked groupings that contain OR producers.

4.9.6 IR availability

IR availability is determined by whether the IRs of a FU can retain the values produced by the being-mapped **FU linked grouping**, without any live value being overwritten. If this can be guaranteed, the FU passes the check.

Section 4.3 explains more about the lifespans of live values, and subsection 4.7.3 explains more about when an IR is available, and when it can be written to.

5. Scheduler

The proposed scheduler method searches for (and produces) an energy efficient schedule for a given application. Discovering what that schedule looks like is an exploratory process, and involves generating multiple schedules.

Implementations are generated based on the schedules, using the implementation generation method discussed in the previous chapter. These consist of PASM code and a configuration, and embody the intended functionality and runtime of the provided schedules exactly. An 'energy efficient' schedule indicates a schedule for which the resulting implementation is energy efficient.

The proposed scheduler aims to find a balance between power dissipation and performance, such that energy consumption is minimized. It does so by generating various schedules and discovering how their differences affect power consumption and performance. For each schedule, an implementation is generated to gain feedback. Further changes to the schedules can be based on that feedback.

A quick side note is in order at this point. Although this chapter discusses the proposed scheduler, this project will not implement nor test it. This project aims to discover whether our proposed method of configuration generation through scheduled based mapping (i.e. the implementation generator) is a viable method of generating configurations (and PASM). Only if that is determined is it sensible to continue the research and discover whether the scheduler could function as desired. The proposed scheduler can not properly be tested without the implementation generator, as it so heavily depends on it.

Section 5.1 explains the format of the schedule. Section 5.2 proposes a method of how the scheduler can approximate the optimal energy efficient schedule.

5.1 Schedule format

The schedules produced by the scheduler are an expansion on CDFGs. They contain the instructions (in target platform format), dependencies, basic blocks, and the control flow between basic blocks. The only additional information it provides it when each instruction is executed (i.e. which basic block and which timeslot), and some specifics regarding internal registers.

This section discusses the format specifics of the schedules. Schedules are implemented in JSON format. The different entities in the schedule are discussed in the subsections below.

5.1.1 Schedule

- List of basic blocks (5.1.2)
- List of IR dependencies (5.1.7)
- List of FIR dependencies (5.1.8)

5.1.2 Basic blocks

- Basic block ID
- First block boolean
- Terminating block boolean

- List of successor basic block IDs
- List of instructions (5.1.3)

The “Basic block ID” indicates the unique ID of this basic block.

The “First block boolean” indicates whether this basic block is the initial block that is to be executed.

The “Terminating block boolean” indicates whether this basic block is a terminating basic block.

The “list of successor basic block IDs” contain the IDs of basic blocks that could potentially be executed after this basic block.

The “list of instructions” is a list of the instructions that are executed during this basic block. The length of the basic block is determined by the timeslots of the first and last instruction.

This method of storing basic blocks necessitates that each possible branching operation is statically recorded. This includes the dynamic branching operations.

5.1.3 Instruction

- Instruction ID
- Timeslot to execute
- Instruction type (5.1.4)
- Generic value
- Producer sets
- Explicit IR indices
- List of LM linked instruction IDs

The “Instruction ID” is the unique ID of this instruction.

The “Timeslot to execute” indicates the timeslot within the basic block to execute. The length of the basic block is one plus the difference between the last and first timeslot values of the instructions it contains.

It was designed such that the timeslot values relate only to the other instructions in the basic block. This allows a developer to make changes to the schedule without the need to consider the timeslot ranges of other basic blocks. However, at the time of writing this has not been tested and is likely not functional yet. Currently, the restriction should be upheld that the timeslot equals the line of code it will appear on in the PASM, relative to the first line of instructions.

The “Instruction type” indicates the instruction type of the instruction.

The “Generic value” element contains generic values (e.g. strings) that need to be added in the PASM code. These include type identifiers (e.g. “BYTE” in ‘lgi BYTE, out1’) and the constant values for branching instructions as well as immediate value producers (e.g. “20” in ‘bcai 20, inX’, and ‘imm 20’ respectively)

The “Producer sets” is a list of OR producer sets. Each OR producer set is specified with a list of IDs. The producer sets are listed in the order of the input operands they produce for.

For instance, '[[1,2],[3]]', for instruction 'sub outX, inY, inZ', means that the value made available on 'inY' needs to originate from the instructions with either ID 1 or 2. The value made available on 'inZ' needs to be produced by the instruction with ID 3.

If the "Producer sets" is not an empty list, then the instruction is an OR consumer.

The "Explicit IR indices" indicate the specific IRs which are explicitly accessed by the instruction. They are used only for PASM generation. They are in the order of how they need to occur in the PASM. This allows the scheduler control over which instruction accesses which IR. This distinguishes instructions that access the special registers of the LSU. This is currently implemented identically for the RF. However, in the future, register allocation of the RF should probably be up to the implementation generator instead.

The "List of LM linked instruction IDs" contains all IDs of instructions that belong to the same LM linked grouping. This indicates which instructions need to access the same range of local memory (i.e. they depend on each other), and therefore need to map to the same LSU.

In the future, this needs to be implemented as a separate entity. That way, the LM linked groupings can also specify which LM dependencies are **not** compatible with each other, or which ranges they need reserved. This prevents that instructions overwrite data that was still depended upon.

Future

In the future it might be possible for the schedules to only specify basic block length, and relative order of instructions. Specific scheduling would then be performed by the implementation generator, and timeslots would no longer appear in the schedules.

5.1.4 Instruction Type

Instruction type consist of the following data.

- Opcode
- List of possible executors (5.1.5)

The "Opcode" is the operation code. It is the string that indicates what instruction is to be executed (e.g. "imm" in 'imm 20').

The "List of possible executors" contains the resources that could potentially execute this instruction type (e.g. "imm" can only execute on the IU).

5.1.5 Possible executor

The possible executor is always in relation to an instruction type. It indicates which FU type could possibly execute the instruction type. Additionally, it specifies which output port the instruction can write to, if it writes values.

This specification enables a distinction to be made for instructions that behave differently depending on the FU type they execute on, or the output port they write to.

For instance, the unbuffered ALU has an unbuffered output port (i.e. without an OR), and a buffered output port (i.e. with an OR). The instructions executed on the ALU can write to both output ports. Writing to the buffered output port takes 1 cycle for the value to be made available on the connection. Writing to the unbuffered output port takes 0 cycles for the value to become available. This is a difference in behavior. Whichever is desired is up to the scheduler.

- FU type (5.1.6)

- Output port indicator

The “FU type” indicates the FU type that can execute the instruction.

The “Output port indicator” indicates which output port can be written to. For the Blocks CGRA, that will be either 0 or 1. If all output ports can be written to, or the instruction does not write a value, then the value is ‘-2’ (arbitrarily chosen out of the negative numbers).

5.1.6 FU type

- FU type name
- config

The “FU type name” indicates the name of the FU type. This needs to match the name of any of the FU types that can be included in the configuration.

The “config” specifies which specific configuration of the FU type this entails. For instance, an ALU with config value 0 is an unbuffered ALU, while an ALU with config value 1 is a buffered ALU. These can generally execute the same instructions, but their behavior will be different.

5.1.7 IR dependency

- Producer ID
- Consumer ID
- IR index

The “Producer ID” indicates the ID of the instruction that is depended upon. This is referred to as the IR producer. It is the instruction that writes to the IR.

The “Consumer ID” indicates the ID of the instruction that depends on the IR producer. This is referred to as the IR consumer. It is the instruction that reads from the IR.

The “IR index” indicates the IR to which the producer writes, and from which the consumer reads. The consumer depends on the value in that IR.

5.1.8 FIR dependency

- Producer ID
- Consumer ID
- List of forwarder IDs
- IR index

The “Producer ID” indicates the ID of the instruction which is depended upon. This is referred to as the FIR producer, and is the instruction that writes to the IR.

The “Consumer ID” indicates the ID of the instruction which depends on the FIR producer. This is referred to as the FIR consumer.

The “List of forwarder IDs” indicates the IDs of instructions that forward the IR with “IR index”, before the FIR consumer reads it. These are referred to as the FIR forwarders.

The “IR index” indicates the IR to which the FIR producer writes, and which is forwarded by the FIR forwarders. The FIR consumer depends on the value in that IR.

5.2 Process

The search for an energy efficient schedule is an ongoing process. However, each ongoing process needs a start. It is sensible for that start to be near the goal (when respect to the search space). This initial schedule is described in subsection 5.2.1.

Before the search process is started, it is prudent to discover some basic things about the application. These are discussed in subsection 5.2.2.

After all that is finished, the scheduler can finally start its search to approximate the optimal energy efficient implementation. How it does so is discussed in subsection 5.2.3.

5.2.1 Initial schedule

The initial schedule aims to be a high performance schedule (i.e. a schedule with few clock cycles). As discussed in section 3.2, the optimal energy efficient schedule is expected to be a high performance schedule.

The initial schedule will consist of ASAP scheduled basic blocks, and pipelined loop nests. Loop nests can not always be pipelined, so whether this is possible will need to be determined in advance. This schedule should be relatively cheap to compute (computation time-wise).

5.2.2 Information gathering schedules

The information gathering schedules aim to discover how often each basic block is executed. That information is used to discover the impact each basic block can have on the total runtime. Preferably, this information is attained from the simulator directly. However, it can be discovered by changing the length of each basic block individually and measuring the change in runtime.

How often basic blocks execute can fully depend on the application input data. It is therefore essential for the used application input data to be representable for general use. It would be beneficial to be able to run the simulator with multiple instances of testing data, or perhaps have the developer specify these numbers himself.

5.2.3 Approximating the optimal solution

In this phase, the scheduler will attempt to approximate the optimal energy efficient solution. This is an iterative process, with each iteration producing a new schedule.

There are two important things to know during this process.

- How often basic blocks execute.
- The required resources per basic block.

How often basic blocks execute is discovered using the information gathering schedules. The required resources per basic block changes as the schedules change. How the required resources change can be estimated based on previous changes, instruction level parallelism (ILP), register usage, and number of dependencies.

ILP indicates the number of FUs per FU type that are absolutely minimally required. Register usage can indicate how many values are live at any given moment, indicating the minimum number of required ORs and IRs, and thereby an approximation of another minimum of the required number of FUs.

The number of dependencies between specific instruction types allows for estimations of the number of connections between FU types. This also allows for an additional estimation on the minimum number of FUs. Unfortunately, the number of connections is very unpredictable, and the estimation will likely be off.

How often a basic block is executed is used to determine the impact on total runtime, if the length of that basic block changes. For instance, if the length of a basic block is doubled, its share in the total runtime would be doubled as well. In the equation for energy, $E = t*P$, this would increase t. However, this could potentially lead to reduced demands on the configuration, which in turn could reduce P. Whichever is dominant (i.e. influences E the most) depends on their percentual change.

Changes in the schedule can have an influence on both t and P. The influence on t is nearly perfectly predictable due to the measurements of how often basic blocks execute. However, changes in P are less predictable.

Changes in P can be estimated based on expected changes in the configuration. For instance, if the contents of a loop nest are duplicated (as to reduce the number of iterations), then the required resources are likely to increase. How many additional resources are needed can be estimated based on ILP, register usage, and the number of dependencies, as described earlier. It is possible that the additionally required resources were already allocated for other basic blocks, and can be reused. This effect can also be predicted.

Based on these estimations, the scheduler can make an estimated guess which changes are most profitable. However, it needs to identify which instructions and basic blocks are most promising to yield beneficial results.

The scheduler could make estimations for each possible change to the schedule, and update those estimations each time a change is applied. This would take a while.

Another method to identify desirable changes, is to look at how often basic blocks are executed. The total runtime is greatly affected by the length of often executed basic blocks. Therefore, resources spent to reduce their lengths will also have a great impact on the overall runtime.

On the opposite end, the basic blocks that are executed only once have a much smaller effect on the total runtime. Any resources allocated solely to parallelize instructions in these basic blocks probably do not reduce t enough to make up for the increase in P.

The scheduler also should balance resource requirements among basic blocks. If resources have been allocated to speed up an often executed loop nest, then these resources could possibly be reused for less often executed basic blocks. Although basic blocks that are only a few times often gain little by parallelizing code, these resources could provide a free performance increase.

Finally, how the instructions are scheduled inside basic blocks also matters quite a bit. This kind of scheduling can be achieved i.a. using force directed scheduling [10]. One of the possible future upgrades of the implementation generator might cause this responsibility to fall to the implementation generator instead.

Other opportunities for energy efficiency can be found in data reuse. It can happen that resource requirements are rather low, except for register usage. The scheduler might be able to make changes to the CDFG to store live values with long lifespans in register files, or possibly even local memory. This could reduce the requirements imposed by register usage, at the cost of one, or more register files.

6. Experimentation

The experiments revolve around the proposed implementation generation method. These experiments determine whether the method is suitable for the proposed overarching tool flow. The goals of the implementation generator were selected with that suitability in mind. Therefore, these experiments test how well the implementation generator performs in regards to its goals.

The implementation generator is based on mapping. However, the mapping process can be blocked due to previous mapping decisions, which requires some instructions to be remapped. This is a lengthy process, and should be shortened as much as possible. This was attempted by trying different initial orders of the OR linked groupings, and trying different reordering methods when too many mapping steps need to be reverted. Which different ordering and reordering methods were tested are discussed in sections 6.2 and 6.3 respectively.

The implementation generator needs to minimize resource allocation. Two separate scoring heuristics were tested, which are discussed in section 6.4. These aim to reduce resource allocation through connection reuse. Finally, section 6.5 discusses how all these methods were tested.

But first, section 6.1 discusses the causes of why the mapping process blocks that are expected to be encountered the most.

6.1 Expected causes of blocked mapping process

There are two causes that are expected to be mainly responsible for blocking the mapping process. The case discussed in subsection 6.1.1 is related to an insufficient amount of input ports to support all incoming connections. The case discussed in subsection 6.1.2 relates to multiple OR linked groupings in a single FU linked grouping.

6.1.1 Producers too wide spread

The first case originates from producers being too wide spread among output ports. This happens when the number of output ports that need to connect to a single FU is larger than the number of input ports a FU has.

For instance, the instructions in a FU linked grouping depend on producers in five unique OR linked groupings. Each of these OR linked groupings writes to a unique OR (and therefore a unique output port). As each FU has only four input ports, it would be impossible to connect to all output ports. No instruction in the FU linked grouping can be mapped until that is resolved.

This case can present itself in two situations. Either the producers are mapped first, or the consumer is. The difference is when the mapping process blocks.

If the producers are mapped first, then the mapping process will block when mapping one of the consumers. No FU will be found that has sufficient input ports to support all the required incoming connections.

If the consumer is mapped first, then the mapping process will block while mapping one of the producers. This situation is always preceded by an overbooking of input ports. The producers need to write to an OR that can be connected to the FU executing the consumer. If all input ports of that FU are taken, producers will only be able to write to ORs with existing connections to the FU. However, this might not always be possible and the mapping process could block.

The problem can be resolved by remapping one or more of the depended upon OR linked groupings, such that they map to at most four unique output ports. This is implemented through backtracking.

6.1.2 Multiple OR linked groupings

This case can occur for FU linked groupings containing multiple OR linked groupings. If the first OR linked grouping of a FU linked grouping is mapped, then the other OR linked groupings are forced to map to the same FU. However, the ORs on that FU are not necessarily mapping options for those other OR linked groupings. This can occur because the ORs in the FU are insufficiently available to reserve for the live values of the OR linked groupings.

Backtracking is required to resolve this problem. One option is to remap the initial OR linked grouping that forced the entire FU linked grouping to that FU. However, the mapping issue can also be caused by other OR linked groupings from the same FU (an example is discussed in subsection 2.2.2, paragraph "Different mapping of the instructions"). Remapping them on the same FU but to other ORs may also resolve the problem.

6.2 Different initial ordering methods

These ordering methods all aim to either avoid backtracking, or reduce the backtracking depth required to solve issues.

Each mapping step maps a OR linked grouping. Therefore, the ordering methods work on at least OR linked grouping granularity.

6.2.1 Top down

The top down ordering method orders OR linked groupings in order of their first executed instructions. The goal is to map producers before their consumers.

However, OR linked groupings contain multiple instructions. These can be scheduled throughout the entire application. Therefore, it is not always possible to map all producers before their consumers.

Per basic block, OR linked groupings are ordered by their first occurring instruction in that basic block. The control flow graph is parsed top down, breadth-first, to process all basic blocks. If an OR linked grouping was added to the order before, it is not added again.

The intended order aims to reduce the number of reservations during any mapping step. Mapping producers before their consumers would facilitate this, because reservations are only created if consumers are mapped before producers.

Reducing reservations could increase the probability of FU and OR reuse, and thus help minimize resource allocation. If a producer with a reserved input port, reuses a connection to reach its consumer, then its reservation is canceled without using an additional input port. The reservation was unnecessary. However, while that unnecessary reservation exists, it can block FU linked groupings to map to that FU.

6.2.2 Top down - FU grouped

This ordering method is quite similar to the "Top down" ordering method, as described in subsection 6.2.1. The difference is the granularity. This ordering method orders FU linked groupings

in order of their first executed instruction.

Per basic block, the FU linked groupings are ordered by their first occurring instruction. Basic blocks are parsed as for the "Top down" ordering method. OR linked groupings within a FU linked grouping have no defined order. If an OR linked grouping was added to the order before, it is not added again.

This ordering method aims to reap some of the benefits of the "Top down" order. Additionally, it aims to reduce the backtracking depth if the mapping process blocks due to the "Multiple OR linked groupings" cause, as described in subsection 6.1.2. That case deals with problems caused by having OR linked groupings in a single FU linked grouping. By mapping them sequentially, the backtracking depth required is reduced (to the number of OR linked groupings in that FU linked grouping).

6.2.3 Top down - Sort by number of dependencies - FUGrouped

This ordering method orders the 'problem cases' first. These are the OR linked groupings that have the most dependency relations (i.e. both depending and depended upon relations). These are the problem cases, because their large number of dependencies requires them to have a large number of connections, both incoming and outgoing. These are extra requirements on both the input ports and output ports of the checked FUs.

The OR linked groupings are first put in "Top down" order, and subsequently ordered by their number of dependencies. The "Top down" order serves as a tie-breaker if an equal amount of dependencies exists for two OR linked groupings. Finally, similar as for "Top down - FU grouped" (as discussed in subsection 6.2.2), the OR linked groupings are grouped by their FU linked groupings.

The final order consists of FU linked groupings that are ordered based on their most problematic OR linked grouping. This again aims to reduce the backtracking depth when FU linked groupings contain multiple OR linked groupings.

Dependency requirements need to be met when the last producer or consumer is mapped (i.e. when the producing output port and consuming FU are both known). By mapping the OR linked groupings with the most dependencies first, satisfying these requirements will fall to OR linked groupings, that (as they are ordered later) have potentially less dependencies to satisfy.

6.2.4 Bottom up

The bottom up methods orders OR linked groupings in order of last-appearance. Opposite to the "Top down" ordering method, this aims to map consumers before their producers, such that the number of reservations is maximized.

The order aims to maximize the benefits of the scoring heuristic described in subsection 6.4.2. This scoring heuristic aims to maximize the reservations that are canceled. Canceling a reservation can only be done by reusing input ports, rather than allocating new ones.

The OR linked groupings are clustered by basic block. The basic blocks are parsed bottom up, breadth-first. Per basic block, the OR linked groupings are collected in bottom up order.

6.2.5 Bottom up - FU grouped

This ordering method is similar to the "Bottom up" ordering method, as described in subsection 6.2.4. The main difference is granularity. This ordering method orders FU linked groupings in

order of last-appearance.

It aims to reap some of the benefits of the "Bottom up" order. Additionally, it aims to reduce the backtracking depth in case the mapping process is blocked due to "Multiple OR linked groupings" cause (which is described in subsection 6.1.2). This is similar as for the ordering method described in subsection 6.2.2.

6.2.6 Bottom up - Sort by number of dependencies - FUGrouped

This ordering method is nearly identical to the ordering method described in subsection 6.2.3. The only difference is that this method uses the "Bottom up" order as a tie-breaker. This tie-breaker only applies when OR linked groupings have an equal amount of dependency relations.

6.3 Different reordering methods

Reordering occurs when the mapping process has reached its maximum backtracking depth. When this maximum depth is reached, the current order, and the blocking OR linked grouping are provided to the reordering algorithm. The blocking OR linked grouping is the grouping that blocked the mapping process and initiated the backtracking algorithm. After the OR linked groupings are reordered, the mapping process restarts from scratch with the new order.

In the subsections below, the different reordering methods are discussed. They all reorder by moving the blocking OR linked grouping to the front of the order. However, there are some minor differences in their specifics. The titles of the subsections indicate what is brought to the front of the order, and in what order.

6.3.1 Blocking OR linked grouping

This reordering method moves the blocking OR linked grouping to the front of the mapping order. As it is now the first OR linked grouping to map, there should be no reason for it to fail its mapping. The only exception is if the OR linked grouping is inherently impossible to map to any FU.

6.3.2 Blocking OR linked grouping - Blocking FU linked grouping

This reordering method moves the blocking OR linked grouping to the front of the mapping order, followed by the remainder of the blocking FU linked grouping (i.e. the FU linked grouping that contains the blocking OR linked grouping).

This can reduce the required backtracking depth, if the OR linked grouping was blocked by the "Multiple OR linked groupings" cause (as described in subsection 6.1.2).

6.3.3 Producers FU linked groupings - Blocked FU linked grouping

This reordering method moves the FU linked groupings of the producers for the blocking OR linked grouping to the front of the mapping order. These are then followed by the blocking FU linked grouping.

The clustering by FU linked groupings reduces the required backtracking depth for the "Multiple OR linked groupings" cause (as described in subsection 6.1.2). Additionally, the producers are mapped before their consumers, which should reduce the number of reservations for these groupings.

6.3.4 Blocked FU linked grouping - Producers FU linked groupings

This reordering method is similar to the reordering method discussed in subsection 6.3.3. However, in this case, the blocking FU linked grouping is mapped first, followed by the FU linked groupings that contain producers for the blocking OR linked grouping.

This leads the blocking OR linked grouping to reserve its input ports for its producers. The aim is to maximize the effectiveness of the scoring heuristic discussed in subsection 6.4.2, which deals with cancelable reservations.

Again, the clustering by FU linked groupings reduces the required backtracking depth for the "Multiple OR linked groupings" cause (as described in subsection 6.1.2).

6.3.5 Producers OR linked groupings - Blocked OR linked grouping

This reordering method moves the OR linked groupings of the producers to the front in the mapping order, followed by the blocking OR linked grouping.

The producers are mapped before their consumers, which can reduce the number of reservations made. Additionally, it decreases the required backtracking depth in case of the "Producers too wide spread" cause, described in subsection 6.1.1.

6.3.6 Blocked OR linked grouping - Producers OR linked groupings

This reordering method moves the blocking OR linked grouping to front of the mapping order, followed by the OR linked groupings of its producers.

The blocking OR linked grouping is mapped before its producers. Therefore, input ports are reserved for all of its producers. This aims to maximize the benefits of scoring heuristic discussed in subsection 6.4.2, which deals with cancelable reservations.

6.4 Scoring heuristics

The scoring heuristics aim to maximize reuse of connections and input ports. There are two separate heuristics tested for this project, which are discussed in the subsections below.

Both heuristics are always applied during the mapping process. The experiments vary which one takes priority over the other. If one scoring heuristic has priority, then the other scoring heuristic is only consulted in case of a tie.

6.4.1 Suitability scoring

This scoring heuristic scores points for every **possible** reuse of a connection. The score the FU and OR pair receives is based on the suitability to support the dependencies.

Points are earned per mapping option for connections that allows dependency satisfaction. For instance, if an OR consumer is being mapped, and a FU has a connection to its producer, then that is worth a point.

Unmapped producers and consumers can also contribute to the score. In this case, a connection needs to exist to a FU that can execute those unmapped producers or consumers. This adds to the suitability of the considered mapping option, as it potentially supports more dependencies.

Subsection 4.9.4 explained that during the mapping process, six different kinds of dependencies

can be encountered (distinguished by mapping state and direction). These contribute differently to the total score.

The paragraphs below explain for each case how points can be earned. The paragraph titles indicate the status of the other party of the dependency (e.g. "Unmapped producer" indicates that a consumer is being mapped, and that it depends on a producer that is not mapped, nor being mapped).

Note that a separate scoring is generated for the FU as a whole, and its individual ORs. However, ORs are only scored if the being mapped OR linked grouping contains OR producers. Each mapping option for OR producers is a pairing of a FU and one of its ORs. The score of that mapping option is the sum of the scores of the FU and the OR. For non OR producers, only the FUs are considered. The score of the mapping options is equal to the score of the FU it represents.

Each of the paragraphs will indicate whether the score applies to the FU or the OR.

Unmapped producer

These points apply to the FU score.

For each dependency relation with an unmapped producer, the FU can earn a point. If an incoming connection exists that originates from a FU that could execute the unmapped producer, then a point is earned. Only one point per unmapped producer and connection can be earned.

Note that this can include a self connection.

Being mapped producer

This score is a subset of the "Being mapped consumer", and will not receive individual points.

Mapped producer

These points apply to the FU score.

Mapped producers have been mapped to an OR. The incoming connection therefore specifically needs to originate from that OR. A FU can earn a point for each mapped producer for which an incoming connection exists to that FU.

Unmapped consumer

These points apply to the OR score.

The consumer is not yet mapped. However, if an OR has connections that lead to the FU type that can execute its consumer, it earns a point. Each connection and consumer can only contribute one point.

Being mapped consumer

These points apply to the OR score.

As these consumers are in the state of being mapped as well, they must be part of the FU linked grouping of the being-mapped OR linked grouping. Therefore, both producer and consumer will always map to the same FU. A self connection is necessary to satisfy this dependency. Therefore, if the considered OR has a self connection, then that specific OR earns a point.

Multiple self connections do not earn multiple points. One self connection is assumed to be sufficient.

Mapped consumer

These points apply to the OR score.

In this case, the FU that executes the consumer is known. ORs earn a point for each unique FU that executes one of the mapped consumers, and that the OR connects to.

6.4.2 Cancelable reservations

This heuristic aims to maximize the number of canceled input port reservations. Canceled reservations do not use input ports. This is a much more 'short term' strategy than the "Suitability scoring" heuristic.

When a consumer maps, it will reserve an input port for all OR linked groupings that contain depended on unmapped producers. If such a depended on OR linked grouping maps to an OR that has a connection to the FU executing the consumer, then its reservation is canceled. No additional input port is used, and the reservation is no longer required. For each dependency that is resolved in this manner, one less input port and connection is required.

This scoring heuristic aims to prevent producers from spreading out among output ports. It is expected to work best when consumers are mapped before their producers.

6.5 Experimentation

The experimentation process is based on existing Blocks CGRA benchmarks. The schedules of these benchmarks were extracted, using a set of custom tools. The resulting schedules are subsequently used by the implementation generator to generate PASM and configurations. The resulting implementation would have the same functionality as the original benchmark. Both the proposed implementation generation method and the resulting configuration will be subject to several measurements.

After the PASM and configuration file have been generated, they are subjected to the same testing environment as the original benchmarks. This tests whether they actually display the same functionality as the original benchmark.

The schedule extraction tools are discussed first in subsection 6.5.1. Following that, the benchmarks are discussed in subsection 6.5.2. Finally, the measurements are discussed in subsection 6.5.3.

6.5.1 Schedule extraction tools

The schedule extraction tools deal with three files: the PASM file, the configuration file and the placing and routing file.

The schedule is extracted from the PASM and configuration file. However, the configuration can already have been placed and routed. The important configuration information will then be retained in a placing and routing file, instead of a configuration file.

Place and routing conversion

If a placing and routing file exists, the first step is to extract the information that is used for the configuration. The placing and routing file consists of FUs and instruction feeds, their positions (of both the FUs and instruction feeds). It also contains the dynamic routing information of the routing hubs.

The ports in the dynamic routing hubs include input and output ports of FUs and instruction feeds, as well as ports that lead to each other. Dynamic connections are established through these dynamic routing hubs, for both data communication and instructions.

The FUs and instruction feeds are listed. By tracing the paths from inputs ports to their origins, the connections between FUs can be extracted.

A new configuration file is generated, that contains all the information. That new configuration file will be used for schedule extraction.

Schedule extraction

The schedule is extracted from the PASM and configuration files. The schedule needs to contain the control flow graph and all data flow graphs of the original application.

The PASM file contains all the instructions, when they are executed, and how they interact with the configuration. The basic blocks can also be extracted from the PASM file, through control flow reconstruction. However, the PASM alone is insufficient for extracting the dependencies between instructions. This can only be extracted through the configuration and how the instructions interact with it.

For instance, instruction 'add out1, in1, in2' shows from which input ports it gets its data from and to which output port it is written. However, the configuration contains the information on which FUs and output ports they connect to. Combining both enables determining the origin of the input data through backtracking through the control flow graph, and allows for the dependencies to be established.

Both the control flow graph and data flow graphs can be created, based on the instructions, their dependencies and the basic blocks.

However, schedule extraction does not work for all PASM. The control flow reconstruction can fail due to dynamic branching (i.e. branching based on an inputted value, rather than an immediate value provided by the instruction). Dependency reconstruction can fail due to dynamic register loading and storing (i.e. instructions which get the index of the register accessed, from one of their inputs). Additionally, multi-core implementation are currently not supported.

6.5.2 Benchmarks

At the time of writing, there are 50 benchmarks for the Blocks CGRA. These benchmarks consist of a PASM file, a configuration file, and some have a place and route file. They implement several different algorithms and many are written with a focus on energy efficiency, or to demonstrate different use cases of the CGRA. The benchmarks are written by a developer of the Blocks CGRA.

Schedule extraction succeeded for 26 of the benchmarks. Below is a list of the benchmarks that were used for the experiments, including whether or not their schedules could be extracted. Only the benchmarks for which a schedule was generated are discussed in the results chapter.

Benchmark name	Schedule extracted
binarization/scalar/	Yes
binarization/scalar_dynamic/	Yes
binarization/tapeout1_dynamic/	Yes
binarization/tapeout2_dynamic/	Yes
binarization/vector4/	Yes
binarization/vector8/	Yes
binarization/vector8_dynamic/	Yes
butterworth/bypass/	No
butterworth/impl5/	No
butterworth/naive/	No
convolution/bypass/	No
convolution/naive/	No
convolution/pipelined/	Yes
convolution/pipelined_dynamic/	Yes
erosion/vector8/	Yes
erosion/vector8_dynamic/	Yes
FFOS/vector8/	Yes
FFOS/vector8_dynamic/	Yes
fft/parallel/	No
fft/parallel_dynamic/	No
fir/naive/	No
fir/parallel/	Yes
fir/parallel_dynamic/	Yes
iir/parallel/	Yes
iir/parallel_dynamic/	Yes
multi-granular/naive/	No
multi-granular/switchbox/	No
MultiProcessor/MP_addition/	No
MultiProcessor/MP_communicationGM/	No
projection/vector8/	Yes
projection/vector8_dynamic/	Yes
SIMD/binarization/	Yes
SIMD/convolution/	No
SIMD/erosion/	No
SIMD/FFOS/	No
SIMD/fft/	No
SIMD/fir/	No
SIMD/fir_shift/	Yes
SIMD/iir/	No
SIMD/projection/	No
tests/tapeout1_count10/	Yes
tests/tapeout2_count10/	Yes
VLIW/binarization/	Yes
VLIW/convolution/	No
VLIW/erosion/	No
VLIW/FFOS/	No
VLIW/fft/	No
VLIW/fir/	Yes
VLIW/iir/	Yes
VLIW/projection/	No

Table 6.1: List of benchmarks

6.5.3 Measurements

During the experiment, multiple different metrics are measured. These are related to the goals of the implementation generator.

The most important metric is whether or not the implementation generation succeeds. Due to the backtracking depth limit and limited number of retries, an implementation generation attempt can be aborted without success. Computation would take too long.

The goal to minimize the number of FUs, requires the FUs to be counted. This measurement can only be taken if the implementation generator succeeds in generating a configuration. The same holds for the the goal to minimize the number of connections. Information regarding both can be retrieved from the resulting configuration, which contains all FUs and connections. Both metrics are compared to the original benchmarks.

The goal to generate the schedule as-is can be tested by subjecting the resulting implementation to the same tests as the original benchmarks. These tests provide specific input data and test for specific output data. The benchmarks pass these tests. If the functionality of the generated implementation matches the original benchmark, then it should pass these tests as well.

The tests are performed using the simulator. The simulator indicate the total runtime of the implementations. The runtime of the implementation is compared to the runtime of the original benchmark.

The computation time of the implementation generator is measured. The implementation generator will be executed from a script. This script stores a timestamp right before, and after the implementation generator is executed. From this, the total computation time can be measured, and compared to the goal set regarding computation time.

7. Results

This chapter discusses the results of the experiments. The goal is to determine if the proposed method is suitable for the proposed overarching tool flow. Additionally, from the results we hope to discover which initial ordering, reordering, and scoring heuristic work best, and how well those work. This will be determined from the number of configurations that are generated successfully, how many of those actually still conform to the intended functionality, the number of allocated resources, and the computation time required to generate the result.

The different combinations of initial ordering and reordering heuristics (or methods) are labeled. These labels are explained in section 7.1. After which, the results are presented. These are divided into separate sections, each discussing the results in regards to one or more goals for the proposed implementation generation method, which are discussed section 4.1. Section 7.2 presents how many benchmarks the implementation generation method was able to generate. Section 7.3 presents how many of the generated benchmarks had the intended functional result. Following that, section 7.4 will discuss how many resources were used in the resulting configurations. These will be compared to the original benchmarks. Finally, section 7.5 will discuss the measured computation times, and makes a comparison between the different cases and benchmarks.

7.1 Labeling

There are a number of different ordering and reordering heuristics, discussed in sections 6.2 and 6.3 respectively. There are six unique heuristics tested for both categories, leading to 36 combinations (72 if scoring heuristics are included). To prevent information clutter, these (re)ordering heuristics and their combinations will be uniquely labeled. The heuristics for scoring are not labeled.

7.1.1 Initial ordering labels

The different initial ordering heuristics determine the starting order of the OR linked groupings. They are labeled as shown in table 7.1.

7.1.2 Reordering labels

The different reordering heuristics determine how the OR linked groupings are reordered. They are labeled as shown in table 7.2.

7.1.3 Combinations

Any combination of the (re)ordering heuristic is labeled by combining their labels. by the label of the initial ordering, an underscore and the label of the reordering heuristic. For instance, the combination of ‘i1’ (i.e. “Top down”) and ‘r1’ (i.e. “Blocking OR linked grouping”) will be ‘i1_r1’. Heuristics regarding scoring are indicated separately.

Initial ordering heuristic	**Code**
Top down	i1
Top down - FU grouped	i2
Top down - Sort by number of dependencies - FUGrouped	i3
Bottom up	i4
Bottom up - FU grouped	i5
Bottom up - Sort by number of dependencies - FUGrouped	i6

Table 7.1: Initial ordering heuristics labels

Reordering heuristic	Code
Blocked OR linked grouping	r1
Blocking OR linked grouping - Blocking FU linked grouping	r2
Producers FU linked groupings - Blocked FU linked grouping	r3
Blocked FU linked grouping - Producers FU linked groupings	r4
Producers OR linked groupings - Blocked OR linked grouping	r5
Blocked OR linked grouping - Producers OR linked groupings	r6

Table 7.2: Reordering heuristics labels

7.2 Generation results

This section will discuss the results in regards to whether the implementation generator can automatically generate implementations.

As stated in section 4.1, the primary purpose of the implementation generator is to automatically generate implementations. To discover how well the implementation generator satisfies this goal, all benchmarks were generated with all combinations of heuristics. There are a total of 26 benchmarks for which a schedule was generated.

For this experiment, the implementation generator has a maximum backtracking depth of 3 mapping steps, and can at most perform 10 reorderings before aborting.

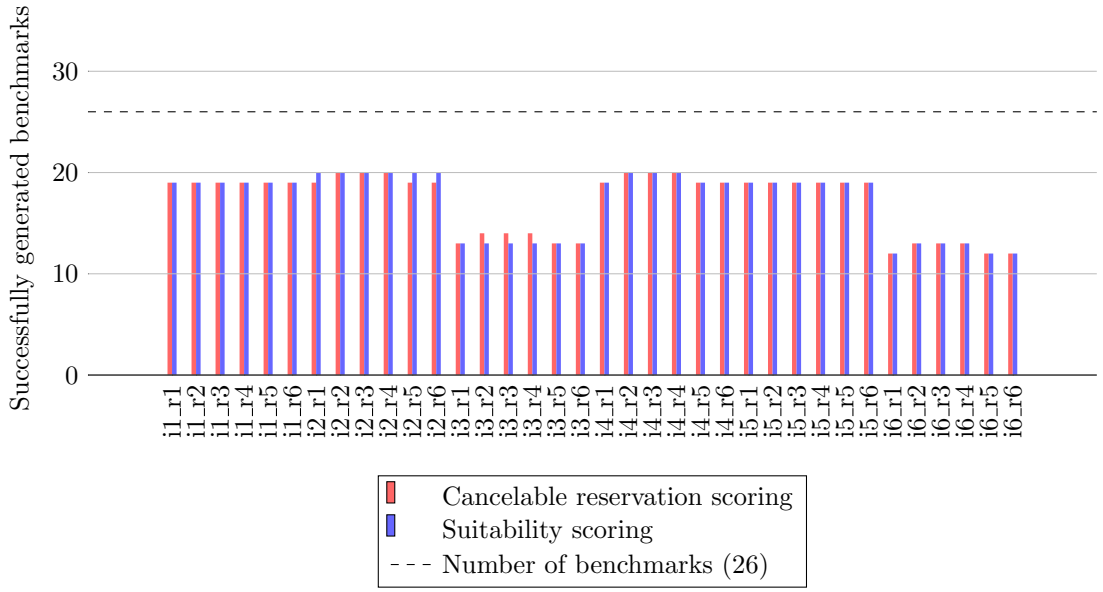


Figure 7.1: Number of generated benchmarks

Figure 7.1 presents the results. From the figure, we can see that at most 20 out of 26 benchmarks are generated for any combination of heuristics. Additionally, we can see that the initial order seems to have the largest impact on the results. Initial ordering heuristic 'i2' combined with 'suitability scoring' seems to perform best, with a 77% success rate. On the other hand, 'i6' seems to perform worst with an average success rate of 48%.

An overall separation is visible. The set of initial ordering heuristics 'i1', 'i2', 'i4', and 'i5' perform better than 'i3' and 'i6'. Both sets are clustered closely together, suggesting there is one significant

difference between both sets.

Initial ordering heuristics 'i3' and 'i6' both sort by the number of dependencies. Their aim is to resolve the problem cases first, while the mapping has not been cluttered yet. This was not beneficial to the final results.

Benchmark name	Times generated
binarization/scalar	72
binarization/scalar_dynamic	72
binarization/tapeout1_dynamic	72
binarization/tapeout2_dynamic	72
binarization/vector4	60
binarization/vector8	60
binarization/vector8_dynamic	48
convolution/pipelined	0
convolution/pipelined_dynamic	0
erosion/vector8	0
erosion/vector8_dynamic	0
FFOS/vector8	0
FFOS/vector8_dynamic	0
fir/parallel	48
fir/parallel_dynamic	27
iir/parallel	72
iir/parallel_dynamic	72
projection/vector8	48
projection/vector8_dynamic	48
SIMD/binarization	72
SIMD/fir_shift	63
tests/tapeout1_count10	72
tests/tapeout2_count10	72
VLIW/binarization	54
VLIW/fir	72
VLIW/iir	60

Table 7.3: Benchmarks generated

The implementation generator tried 72 unique combinations of heuristics, and in effect attempted to generate each benchmark 72 times. Some of them succeeded to generate each time, while others failed to generate even once. Table 7.3 shows how many times each benchmark succeeded to generate. Some benchmarks were never successfully generated. The benchmarks that always failed to generate did so for different reasons. These were investigated as to why they would not generate for the 'i1_r1', with 'suitability score' heuristics.

One of the reasons the "erosion/vector8" benchmark failed, is because its producers were too wide spread (as discussed in subsection 6.1.1). This is the case where the producers were mapped first and the process blocked while mapping the consumer. The FU linked grouping of the consumer, in a specific case, depended on producers in twelve unique OR linked groupings. Five of those were already mapped to unique output ports, which exceeded to total number of input ports. The FU linked grouping of the consumer could therefore find no FU to support all its dependencies.

The "convolution", and "FFOS" benchmarks, as well as the "erosion/vector8_dynamic" benchmark also encountered failures due to producers being too wide spread. However, here the case occurred that the consumer was mapped first. While mapping a producer, it could not find an OR

that could connect to input ports at all of its consumers. The FUs that executed the consumers already had used (or reserved) too many input ports. The mapping process failed, because the backtracking depth was insufficient to remap other producers of the consumer.

7.3 Generation functionality and runtime results

This section will discuss the results in regards to whether the generated implementations conform to the intended functionality and runtime.

The goal discussed in subsection 4.1.1 states that the generated implementations need to execute exactly according to the schedule they were generated for. This pertains to scheduling and functionality. If the implementation generator meets this goal, then the generated implementation displays the exact same behavior as the original benchmark, and has the same runtime.

Subsection 7.3.1 discusses the results regarding functionality comparisons, and subsection 7.3.2 discusses the results regarding the runtime comparison.

Functionality and runtime should not be influenced by the different heuristics. Still, all combinations of heuristics have been tested, to increase the probability of encountering varied and unique cases. Therefore, each benchmark is generated for each of the unique 72 combinations of the heuristics.

7.3.1 Functional comparison

The functionality was tested using the testing data of the original benchmarks. The testing data provide the implementations with input data and specifically expected output data, which the resulting output can be compared with. If the generated implementations have the same functionality as the original benchmarks, then like the benchmarks, they should create the expected output data.

For this experiment, the implementation generator has a maximum backtracking depth of 3 mapping steps, and can at most perform 10 reorderings before aborting the mapping process.

Figure 7.2 shows how many implementations were generated per benchmark, and how much of those were functionally correct (green) or incorrect (red). We can see that a number of the generated implementations do not match the original functionality.

A total of 1236 implementations were generated, of which 1023 were functionally correct. We can see that whether functional discrepancies occurred largely depends on which benchmark was implemented. Only implementations of the "SIMD/binarization" benchmark experienced influence from the heuristics.

Although the functional correctness of the generated implementations depends on which benchmark is generated, figure 7.3 shows a comparison between the heuristics combinations. From this we can see that the initial ordering methods 'i1' and 'i2' generate the most functionally correct implementations. Both succeed to generate a functionally correct implementation for 62% of the benchmarks. 23% of the implementations failed to generate, and 15% of the generated implementations were functionally incorrect. Their success rate likely originates from being able to generate the benchmarks that lead to functional correct results.

Many of the functional discrepancies originate from the simulator refusing to run the generated implementation. In these cases, the generated configuration or PASM did not meet the requirements of the simulator. This indicates that the implementation generator still requires additional work to incorporate those requirements. The following paragraphs discuss the discovered causes

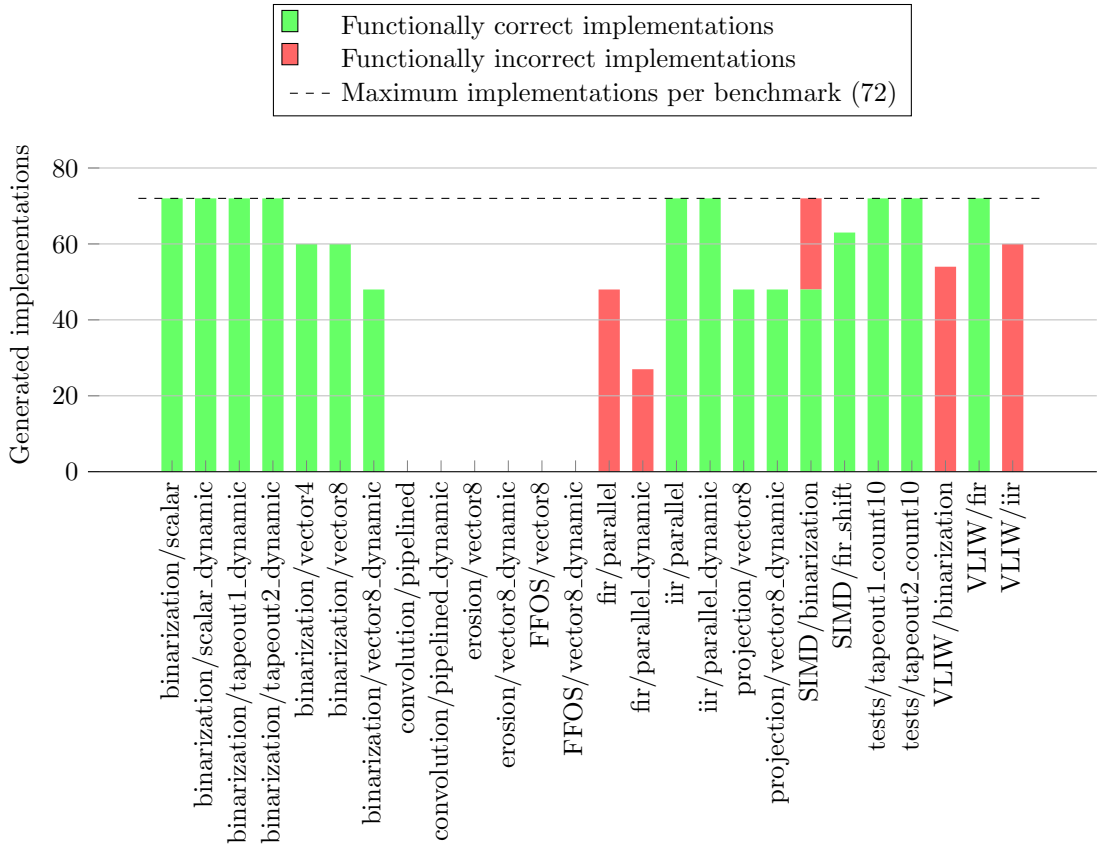


Figure 7.2: Functional correctness of generated implementations

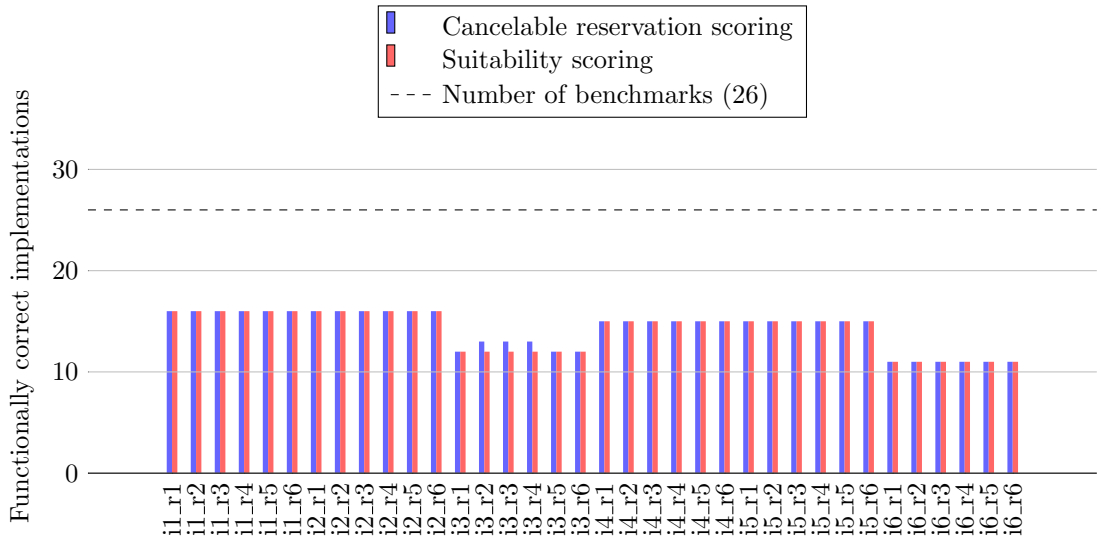


Figure 7.3: Functionally correct implementations per heuristics combination

for functional deviation and/or why the simulator considered an implementation invalid.

Looping unbuffered ALU

One of the reasons why the simulator refused to run an implementation was found in the unbuffered ALU. For such an unbuffered ALU, connecting the unbuffered output port to one of its own input ports is not allowed. This makes sense because results on the unbuffered output require 0 cycles to compute, and can be used by instructions executing in the same cycle. The result could feed back into the input port, which is read to calculate that result.

Benchmarks affected by this: “fir/parallel”, “VLIW/iir”.

Dummy instructions

Some instructions depend on the default value of an OR. That is the value the OR has at start up, before any value is written to it. These ORs needed to be reserved by the proposed generation method, as this is a dependency.

This was solved by introducing dummy instructions. These dummy instructions have a blank opcode (which defaults to the commonly shared ‘nop’ instruction). These dummy instructions are considered OR producers. They are scheduled at the very first time slot of the schedule and can execute on pretty much any FU (except RFs, as their outputted values depend on IRs instead of ORs). One dummy instruction exists for each default-value-dependency. By exploiting the scoring heuristics, they aim to utilize FUs that are already connected to the input ports of the FU executing the consumer. These dummy instructions created some problems.

The implementation generator sometimes creates ‘non-used FU’. These are allocated for the dummy instructions to map to, and are never used otherwise. These FUs have no incoming connections, which is not allowed by the simulator. They can be removed after PASM generation completes. The input ports reserved for them would return to an unconnected state, which also provides the correct default value.
Benchmarks affected by this: “SIMD/binarization”. A notable pattern for the “SIMD/binarization” benchmark, was that this always happened for “i4” and “i5” of the initial ordering heuristics, which employ a “Bottom up” initial order.

Another issue with dummy instructions originates from the blank opcode. The IU has a ‘nopi’ instruction rather than a ‘nop’ instruction. The blank opcode was chosen to deal with this difference, however, this seemed to be interpreted as ‘nop’ on the IU as well. Benchmarks affected by this: “fir/parallel”, “fir/parallel_dynamic”.

Both cases can be resolved. However, it might be preferable to find an alternative solution to the default-value-dependencies.

Inexplicable output data

The “fir/parallel” and “fir/parallel_dynamic” benchmarks had some instances where the generated implementation wrote an additional four bytes of data to the global memory. This can be achieved with a single write operation. These four bytes were the only difference between the resulting output data and the expected output data. The runtime (in cycles) of these implementations matched the runtime of the original benchmark exactly. Additionally, an equal amount of instructions were executed.

A similar case occurred for the “VLIW/binarization” benchmark. The output data indicates some bits are marked as undefined for the last four bytes (out of 8000 bytes of output data). This

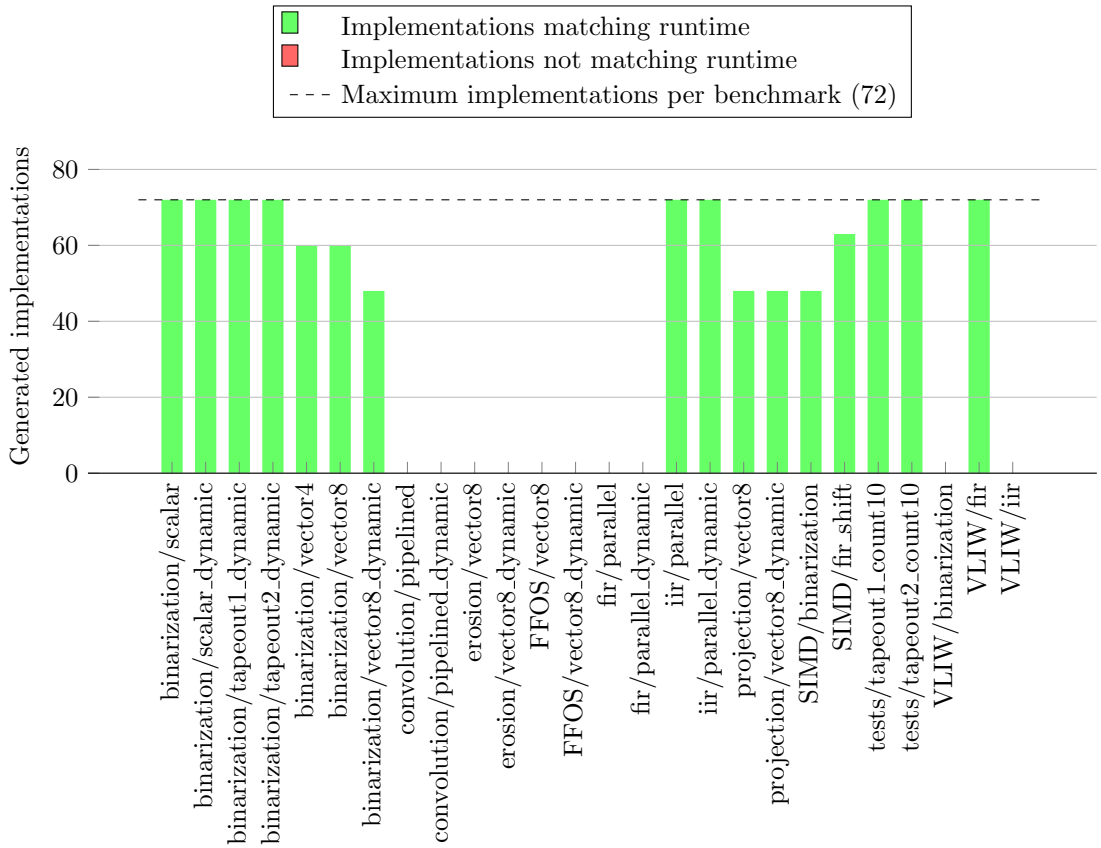


Figure 7.4: Implementations with runtimes matching their original benchmark

suggests a masked write occurred. Again, all other data is correct, and the runtime is identical to that of the original benchmark.

The "VLIW/iir" benchmark is the worst performing benchmark in terms of conforming to functionality. It is often not accepted as a valid implementation by the simulator. However, the instances that are valid produce no output data at all. The runtime, the data flow graph, and addressing of the output data are identical to that of the original benchmark.

I have yet to find an explanation for these discrepancies. The output data never shows deviating bytes, but rather a lack or excess of data.

7.3.2 Runtime comparison

The runtimes of the generated implementation were compared to the runtimes of the original benchmarks.

The original benchmarks were used to generate schedules. Those schedules were in turn used for the implementation generator. Therefore, the resulting implementations should have the same basic block lengths as the original benchmarks.

The Blocks simulator can measure the runtime for any simulated run. The generated implementations were provided with the same input data as the original benchmarks were. Therefore, their runtimes should be identical.

Figure 7.4 shows whether the functionally correct implementations match the runtime of the original benchmark (green) or not (red). All functionally correct implementations match the runtime of their respective original benchmarks.

The figure only includes results that behave functionally identical to the original benchmark. Many of the implementations that deviated were in fact rejected by the simulator and did not have an actual runtime to compare to. Others did have the correct runtime but had inexplicable output data.

Compared to the total 1872 attempts to generate an implementation, 1023 resulted in a functionally correct implementation with a correct runtime. That is a 55% success rate, for a maximum backtracking depth of only 3 and 10 allowed retries per generation attempt.

7.4 Generation resources results

This section discusses how the implementation generator performed in regards to resource allocation. These results include all generated implementations, including the non-functional ones. The resources are based on the resulting configuration, and measurement does not depend on whether the simulator considered them valid.

For these measurements, each benchmark was also generated with each combination of heuristics. Including the non-functional implementations, this resulted in 1236 generated implementations.

The resources are divided into FUs and connections. These will be discussed in subsections 7.4.1 and 7.4.2 respectively.

The number of instruction feeds will always be equal to the number of FUs, minus a small number of IUs (these do not have instruction feeds). As instruction feed sharing was out of scope, this was expected and will not be discussed further in this thesis.

7.4.1 Number of FUs

First, the results are given per benchmark. Following that, the results of the heuristic combinations are compared to each other.

Benchmarks

Figure 7.5 shows the least (green), most (red), and average number of FUs (orange) compared to the number of FUs in the original benchmarks (blue). We can see that the implementation generator performs poorly on minimizing the number of FUs. Only for a few cases does its best results approximate the number of FUs in the original benchmarks.

Figure 7.6 shows the percentage-wise increase in number of FUs compared to the original benchmark. The average increase over all benchmarks has also been depicted per category (i.e. least, most, and average) by the horizontal lines. The least overall increase is 32%, the average overall increase is 51% and the worst is 69%.

The figure clearly shows that the “projection” benchmarks yielded the worst results. After closer inspection of the “projection” results, the ALUs and LSUs seem severely underutilized. A number of LSUs were utilized solely for pass instructions, which can be executed on any FU.

Studying the generation logs of some of the implementations revealed the causes for FUs to fail mapping checks. If all FUs fail their mapping check during a mapping step, an additional FU is

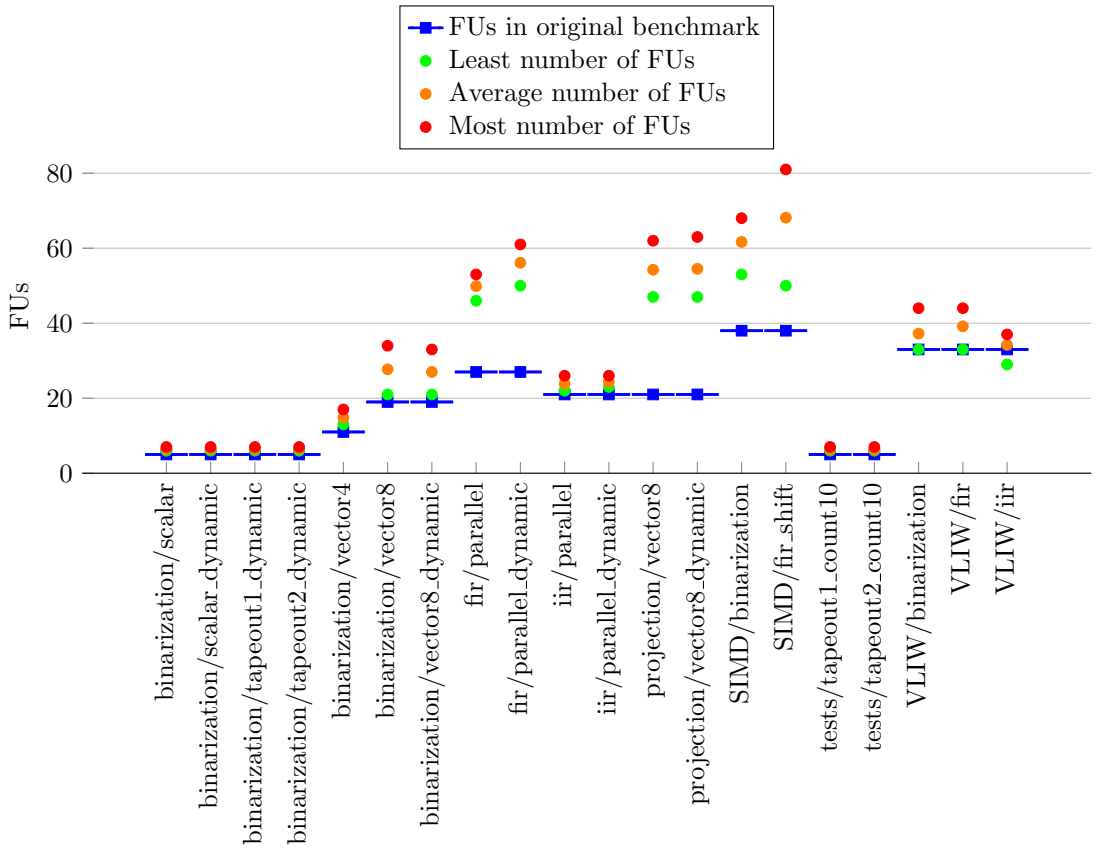


Figure 7.5: Comparison of the number of FUs per implementation

allocated. The numbers are shown in table 7.4.

We can see that the number one cause of FUs failing to pass the mapping checks, is because at least one OR linked grouping of a FU linked grouping is already mapped on another FU. This suggests that many of the FU linked groupings of the benchmark, consist of more than one OR linked grouping.

Another notable cause are the insufficient number of input ports. This suggests that a large number of FU linked groupings depends on the values of a large amount of OR linked groupings. This is resolved by overbooking input ports on a FU with all input ports available. Often this leads to allocating a new FU, and could explain the significant increase in FUs. The pass instructions mapped to the LSUs would require one input port, which would remove those FUs from consideration for overbooking input ports.

Heuristic combinations

We have determined that the implementation generator performs below expectations in regards to minimizing the number of FUs. However, the influence of the heuristics is yet undetermined.

Figure 7.7 shows a comparison of the heuristic combinations. Each implementation is compared to the smallest (i.e. the "Least number of FUs" in figure 7.5) generated implementation of that benchmark, per heuristics combination. Each point represents an average ratio of how much the implementations (per heuristics combination) deviate from the smallest implementation per benchmark. Each point in the figure is calculated using equation 7.1, where B_{HC} is the set of

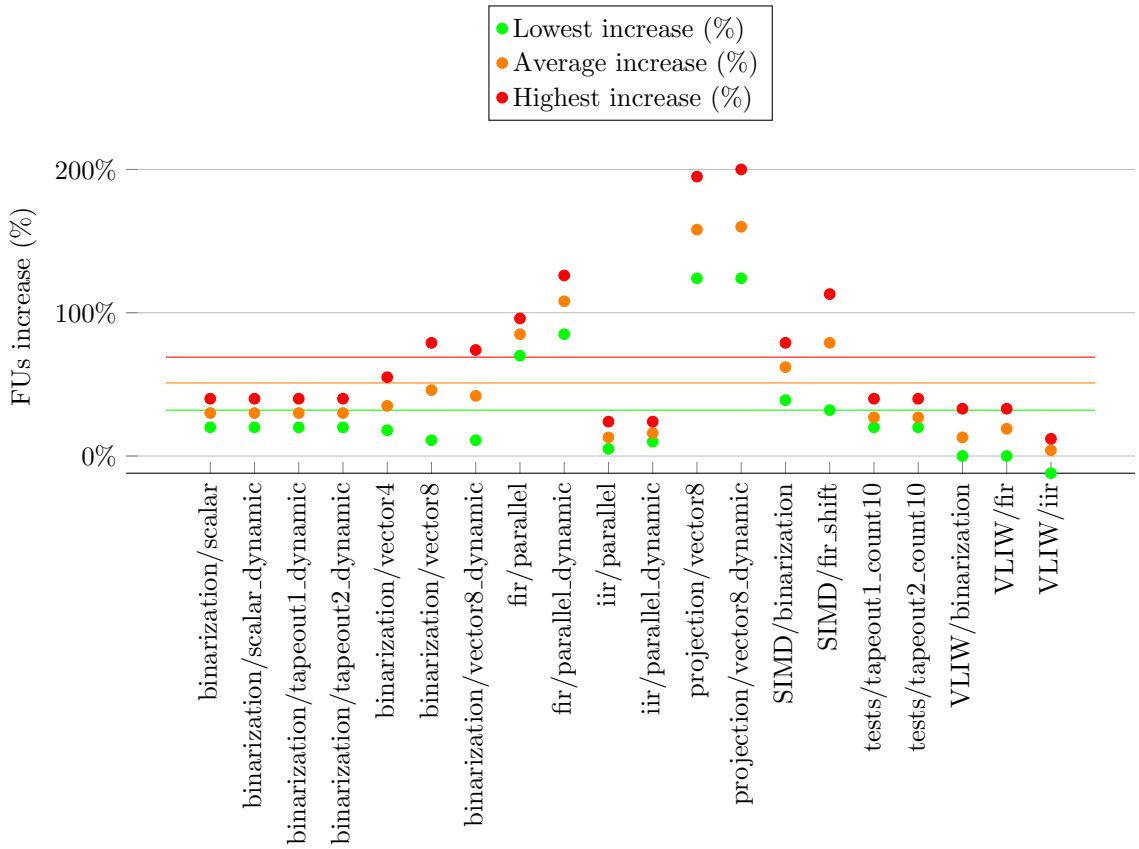


Figure 7.6: Increase in number of FUs, compared to original benchmark

benchmarks for which implementations are generated by heuristics combination HC, and $F_{b,HC}$ is the number of FUs in the resulting implementation when generating for benchmark b using heuristics combination HC, and $F_{b,min}$ is the number of FUs in the smallest implementation generated for benchmark b.

$$\frac{\sum_{b \in B_{HC}} \frac{F_{b,HC}}{F_{b,min}}}{|B_{HC}|} \tag{7.1}$$

From the figure, we can see that particularly the 'i2' initial ordering heuristic seems to be responsible for the best results. The 'i2' label represents the "Top down - FU grouped" initial ordering heuristic. This is also the heuristic that has the best success rate at generating implementations in general. Particularly the 'r2', 'r3', and 'r4' reordering heuristics combined with "Cancelable reservation" scoring seem to perform well, although the difference is minimal. As these heuristic combinations have a value close to 1, this suggests that their results closely approximate the best possible results.

The most negative influence seems to be the 'i4' initial ordering heuristic, which is the "Bottom up" initial order. This makes sense, as it aims to maximize the number of input port reservations. This causes FUs to fail mapping checks, because insufficient input ports remain unreserved, which in turn can increase the number FUs allocated. This can also be seen in table 7.4, where a significant difference in mapping check failures due to "Insufficient available input ports" can be seen between 'i2' and 'i4'.

Ordering heuristic combination	i1_r1	i2_r1	i4_r1	i5_r1
FU linked grouping mapped to different FU	3049	2121	3186	1494
FU execution slot not reservable	438	438	438	438
IRs are not reservable	16	16	16	16
All ORs are not reservable	6	8	79	41
Insufficient available input ports	802	676	2502	2607
Total:	4311	3259	6221	4596

Table 7.4: FU Mapping check fails for "projection/vector8" benchmark; Suitability scoring heuristic

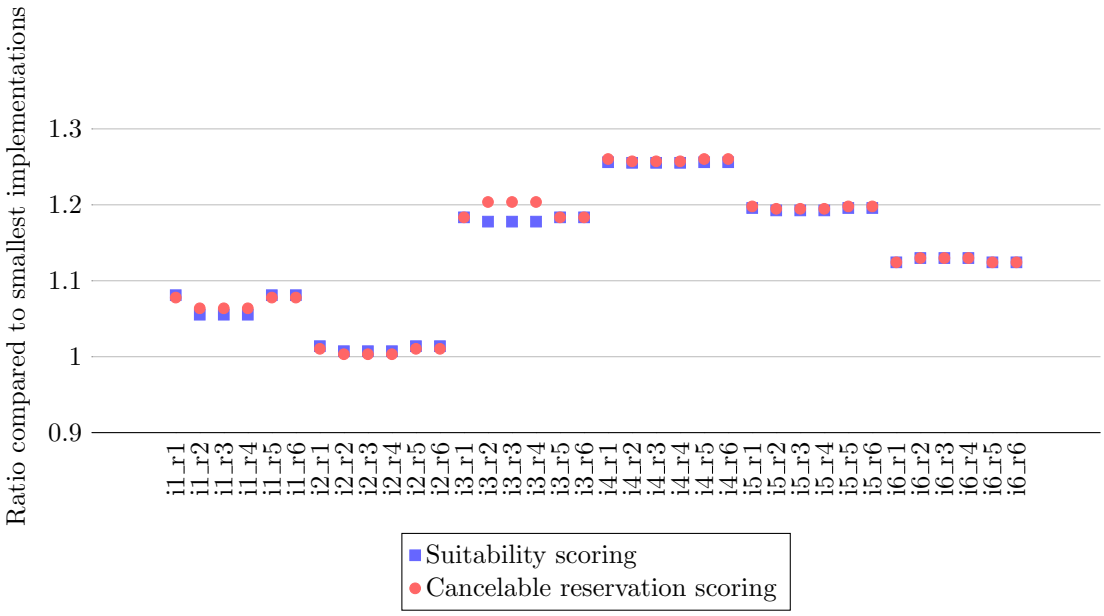


Figure 7.7: Heuristic results compared to best generated implementations (FUs)

It is notable that the "Cancelable reservation scoring" heuristic did not have a positive effect on 'i4' and 'i5' relative to "Suitability scoring". These initial ordering methods were designed to maximize its benefits, but failed to have any significant influence on its effect.

7.4.2 Number of connections

The implementation generator seems to perform poorly for connection allocation as well. This makes sense, as connection reusability drops due the increased number of FUs.

Benchmarks

Figure 7.8 shows the least (green), most (red), and average number of connections (orange) in the generated implementations, compared to the number of connections in the original benchmarks (blue). The implementation generator performs poorly in comparison. However, it has allocated fewer connections than the "iir/parallel" benchmark, and especially fewer than the three "VLIW" benchmarks.

The "VLIW" benchmarks are all implemented using the same general purpose (VLIW-CPU) configuration. Not all FUs and connections of this general purpose configuration will be used for each of the benchmark applications. The generated configurations only supports the application

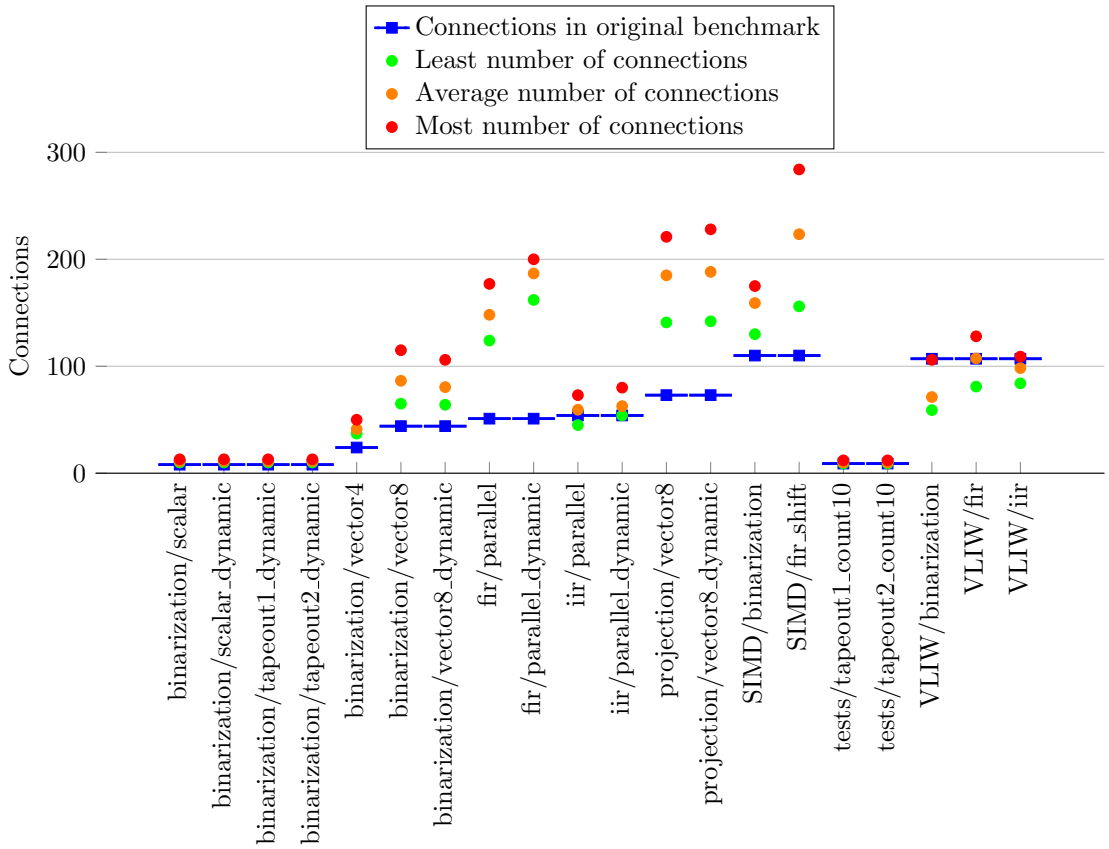


Figure 7.8: Comparison of the number of Connections

of one benchmark, and could therefore reduce the number of connections.

I cannot explain the "iir/parallel" benchmark. Perhaps the proposed implementation generation method managed to beat the human developer solely for this case.

Figure 7.9 shows the percentual increase in the number of connections, compared to the original benchmarks. The horizontal lines show the averages of these increases for all benchmarks, for the best results (green), average results (orange), and worst results (red). Respectively, these overall average increases are 37%, 67%, and 100%.

Especially benchmark "fir/parallel_dynamic" has poor results. The generation log revealed that many mapping checks failed due to insufficient input ports being available, suggesting many FUs were allocated due to this. This is a consequence of the high number of connections required, but could also be a cause for additional connections to be allocated. As the OR linked groupings spread out over several FUs, the reusability of the existing connections reduces. Therefore additional connections are required.

Heuristic combinations

This paragraph discusses which heuristic combination performed best.

Figure 7.10 shows a comparison of the heuristic combinations, in terms of connections. The approach is identical to that of figure 7.7, except that it considers connections instead of FUs. The results of the heuristic combinations are compared to the best generated implementations,

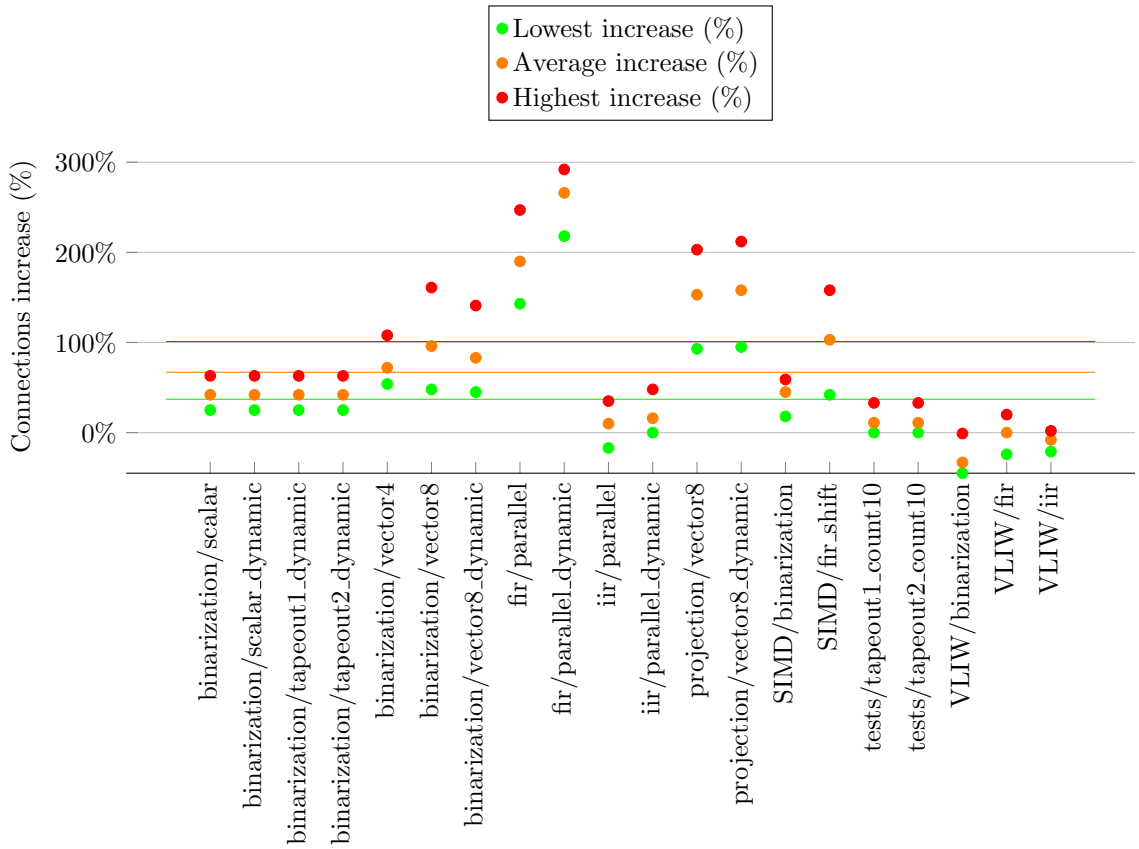


Figure 7.9: Increase in number of connections, compared to original benchmark

in terms of connection minimization. Each point represents how well a particular combination of heuristics performed, compared to the absolute optimal that was achieved by the implementation generator.

Each point in the figure is calculated using equation 7.2, where B_{HC} is the set of benchmarks for which implementations are generated by heuristics combination HC, and $C_{b,HC}$ is the number of connections in the resulting implementation when generating for benchmark b using heuristics combination HC, and $C_{b,min}$ is number of connections in the implementation containing the least connections generated for benchmark b.

$$\frac{\sum_{b \in B_{HC}} \frac{C_{b,HC}}{C_{b,min}}}{|B_{HC}|} \tag{7.2}$$

When comparing figure 7.10 to figure 7.7, we can see that the influence of the heuristics combinations on the connections and FUs is very similar. Therefore, it is not surprising that initial ordering heuristic 'i2' seems to have the most positive effect.

Particularly the "Top down" initial order seem to have beneficial effects for the number of connections. These are the orders that minimize input port reservation, and allow more FU linked groupings to map to the same FU without being hindered by over-zealously reserved input ports.

We can only speculate why the coarser granularity of 'i2' seems to perform better than 'i1'.

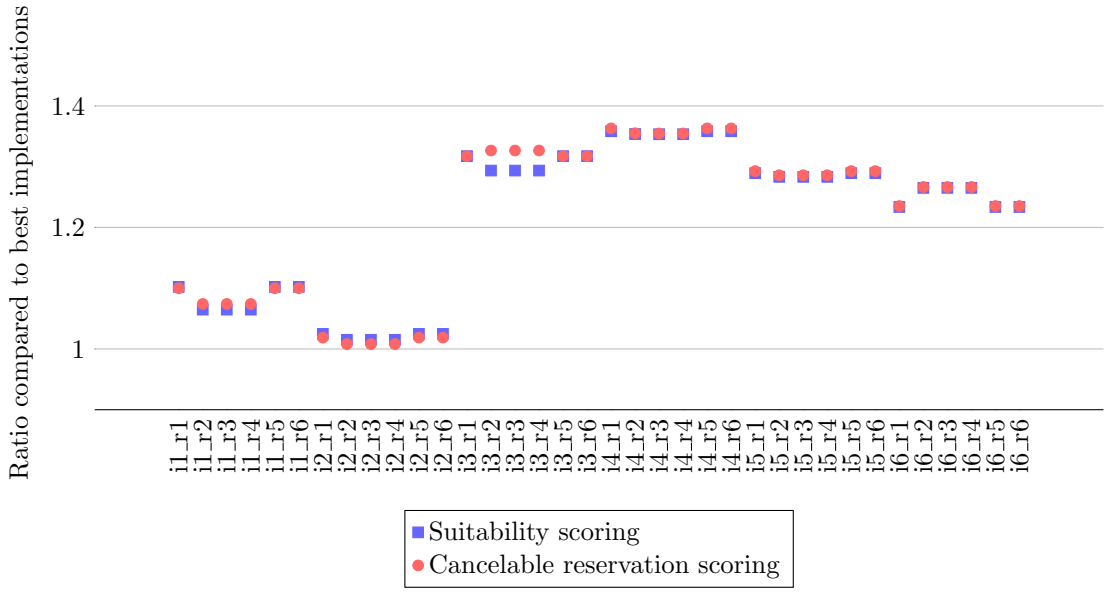


Figure 7.10: Heuristic results compared to best generated implementations (Connections)

The order of 'i2' maps OR linked groupings clustered by FU linked grouping. Only the first OR linked grouping of the FU linked grouping reserves input ports. By mapping the remainder of the FU linked grouping, no additional input ports are reserved. However, by mapping the remainder, some unnecessary reservations might be removed. This enables more FU linked groupings to map to the same FU and encourage connection reuse.

7.5 Generation computation time results

This section discusses how the implementation generator performed in terms of computation time. The computation time is measured in seconds. Like for the other experiments, each benchmark was generated for each heuristics combination, with a maximum backtracking depth of 3 and a maximum of 10 reorderings before the implementation is aborted.

Figure 7.11 shows the best (green), average (orange), and worst (red) computation time that was required to generate implementations, per benchmark. Only the implementation generator executions that succeeded in generating an implementation are included in the data for this figure. It is notable that the best results all finish within 10 seconds, and on average finishes within 2.6 seconds.

When looking at the results, the "SIMD/fir_shift" benchmark seems to take the longest time to compute on average. This was also the benchmark for which the most FUs were generated on average. When comparing the computation time per benchmark in figure 7.11 to the number of FUs in figure 7.5, there are apparent similarities.

Figure 7.12 shows a comparison between the number of FUs and the computation time for all produced implementations. An exponential trend can be observed between the number of FUs and the computation time.

Figure 7.13 shows a comparison of the heuristic combinations. Per heuristics combination, the computation time required for each of the implementations is compared to the shortest computation time (i.e. the "Least computation time(s)" in figure 7.11) in which the related benchmark

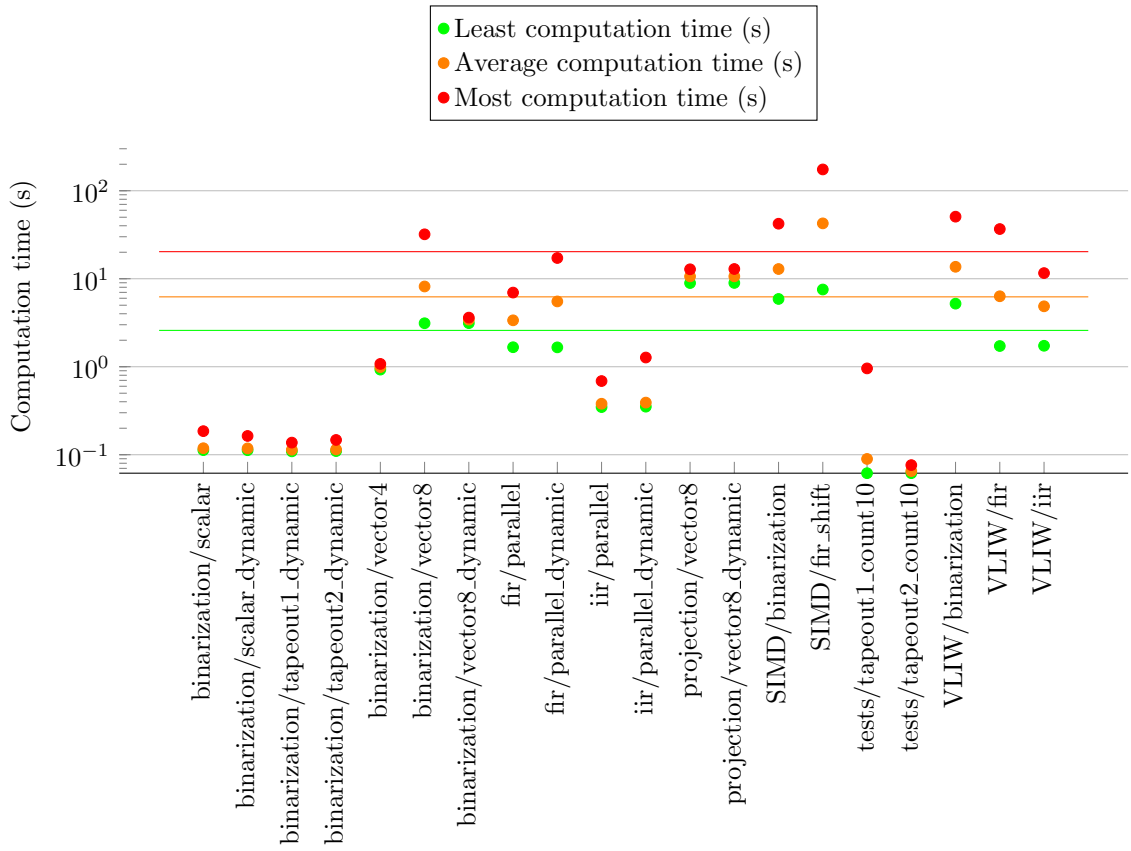


Figure 7.11: Comparison of computation time per benchmark

was ever generated.

Each point in the figure is calculated using equation 7.3, where B_{HC} is the set of benchmarks for which implementations are generated by heuristics combination HC, where $t_{b,HC}$ is the measured computation time (in seconds) when generating an implementation for benchmark b using heuristics combination HC, and $t_{b,min}$ indicates the shortest computation time in which an implementation was generated for benchmark b.

$$\frac{\sum_{b \in B_{HC}} \frac{t_{b,HC}}{t_{b,min}}}{|B_{HC}|} \tag{7.3}$$

From this, we can see that particularly the 'i4' and 'i5' initial ordering heuristics seem to compute their results fast. These are the "Bottom Up" initial orders (where 'i5' does so by FU linked grouping granularity). Ordering the instructions bottom up should maximize the input port reservations at any given cycle. Despite this having an adverse effect on the number of connections and FUs, it does seem to have a positive effect on computation time. This seems contradictory to the trend seen in figure 7.12.

However, 'i2' is not much worse compared to 'i4' and 'i5'. This is notable because it is the best performing heuristic for the other categories (i.e. number of FUs, connections, as well as number of generated functionally correct implementations). Overall, this makes it the best heuristic to choose for initial ordering. Particularly when combined with 'suitability scoring' and 'r2',

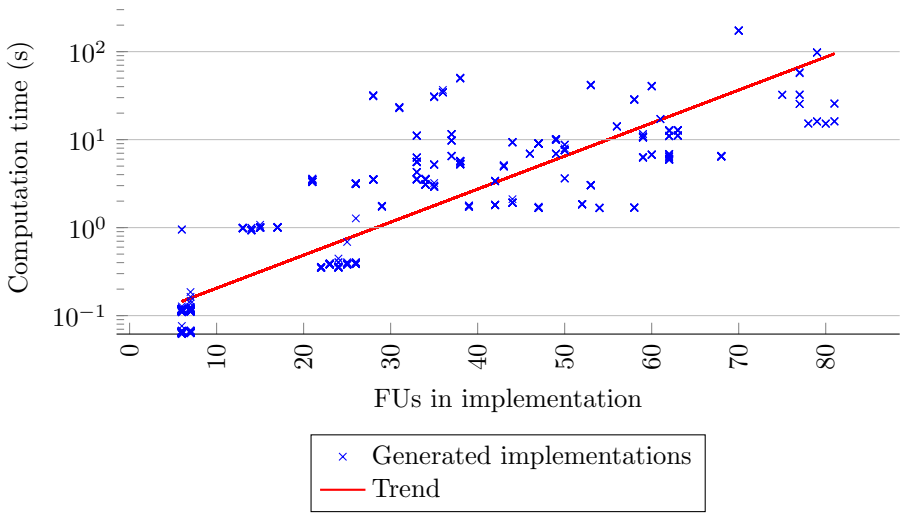


Figure 7.12: FUs compared with computation time

'r3', or 'r4' reordering methods. Notable is that these reordering methods all work on FU linked grouping granularity, similar to 'i2'.

When running the implementation generator with 'i2' (to achieve the best overall results), one implementation can be generated each four seconds on average. The worst case computation time for 'i2' was 42.2 seconds, for the "SIMD/binarization" benchmark.

In contrast to the other metrics, the influence of the reordering and scoring heuristics on computation time is more outspoken. One can especially see this for 'i1', where 'r1', 'r5' and 'r6' perform relatively poorly.

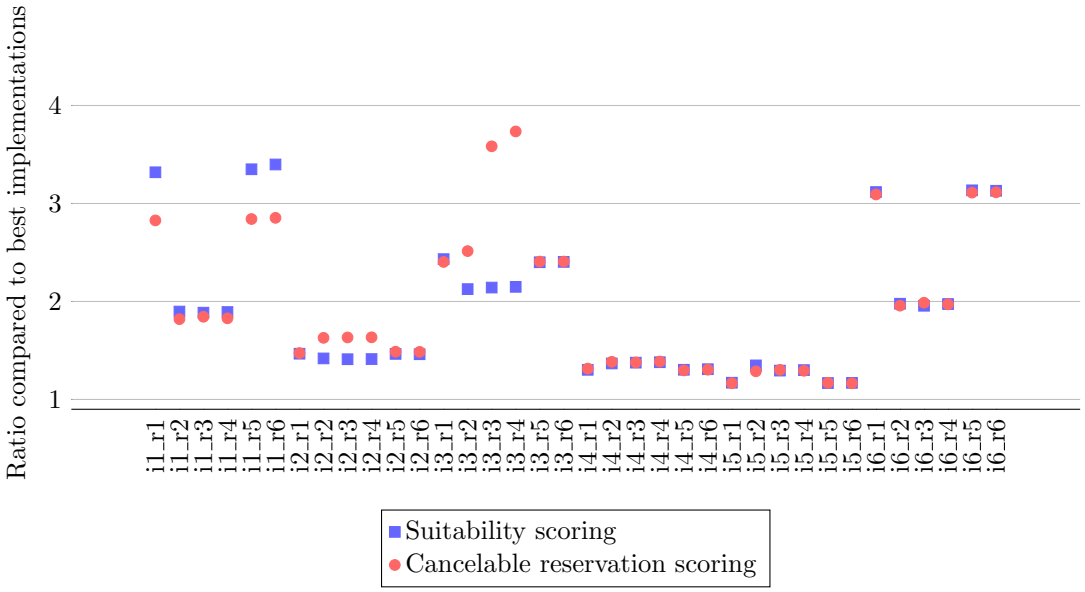


Figure 7.13: Heuristic results compared to fastest generated implementations (Computation time)

8. Conclusion

The proposed method of generating implementations presented in this thesis can successfully generate CGRA configurations and code for schedules containing CGRA applications. It does so by mapping its instructions, dependencies, and values to resources, while adhering to the schedule. Grouping instructions by resource requirements posed by the dependencies, enables simultaneous mapping of small clusters of instructions. By allocating or reusing resources to assert satisfaction of data dependency requirements, the mapping process creates functionally correct implementations.

The mapping process requires backtracking to resolve emergent mapping problems originating from preceding mapping steps. Backtracking can add a significant worst-case time complexity penalty. The computation time grows exponentially as the backtracking depth increases. Limiting the backtracking depth to a maximum of three mapping steps resulted in a computation time that ranges from at most three minutes to less than a second, while still succeeding to generate functionally correct implementations for 55% of all test cases.

The order in which instruction clusters are mapped can increase the probability to avoid mapping problems, minimize the backtracking time penalty, and reduce resource allocation. Clustering instructions that require a shared function unit, and ordering those clusters in execution order of their earliest executing instruction, showed the best overall results. Generating the test cases using this order resulted in functional correct implementations for 62%, with an average of four seconds computation time per implementation, ranging from just over 42 seconds to less than a second.

The proposed method currently does not meet its goal of generating configurations containing minimum resources. Compared to expert developer implementations, the generation process allocated 32% additional function units and 37% additional connections on average. However, the proposed method does satisfy the requirement to compute implementations quickly, as well as the requirement that the runtime matches what is dictated by the schedule exactly. The generating process therefore only needs more work in regards to efficient resource usage, before it is suitable for use in energy efficiency searching methods. The techniques proposed in chapter 9 suggest how the proposed method can be improved.

The proposed method as-is, is suitable as a stand-alone tool to quickly generate CGRA configurations and code for provided schedules. If energy consumption or resource limitations are a concern, then the resulting implementations would need to be refined further by developers. Using the proposed method could still reduce development time in this case.

9. FutureWork

The proposed implementation generation method as is does not meet the requirements of the proposed searching method. However, many improvements are yet possible that might change that.

The initial problem that should be addressed is the success rate of generating configurations. Possible solutions are discussed in sections 9.1 and 9.2. Additionally, a solution to reduce the number of resources in the generated configurations is discussed in section 9.3.

9.1 Smart backtracking

Backtracking aims to solve mapping problems caused by preceding mapping steps. By reverting to previous mapping steps, the mapping process can choose an alternative mapping option. Using this alternative mapping, the subsequent mapping steps are tried again. As discussed in section 4.8, the recursive and repetitive nature of the backtracking process results in a dominating worst case time complexity for the mapping process. For each altered mapping step, all mapping options of all subsequent mapping steps are retried unless the problem is resolved. If the backtracking depth is left unlimited, the mapping process will always find a way to resolve any mapping problem, given that the provided schedule is not impossible. However, it would simply take too long to compute.

A more intelligent backtracking implementation considers the possible influences of each mapping step. By discovering the source of a mapping problem, the backtracking process can be provided with a goal. Using this goal it can consider whether mapping steps are worth altering (and if so, how they should be altered), instead of trying all combinations.

For instance, the mapping process is blocked because a FU lacks an available input port. The backtracking proceeds with the goal to free up one input port of that FU. It can ignore all mapping steps that have no relation to that FU. If it finds an instruction that writes to that FU, it will attempt to remap its outputted value to an output port that is already connected to that FU. Once the problem is resolved, subsequent steps can be tried.

This changes the average case time complexity of backtracking. It no longer tries all mapping options for all mapping steps, but only some options for a select few mapping steps.

9.2 Graph coloring

As discussed in section 4.5, instructions can be clustered into groups based on dependency requirements. To satisfy dependencies, all instructions in that group must share a specific resource.

Each of these groupings requires specific resources which they need to reserve. The periods when resources are reserved can be determined beforehand, due to the provided schedule. If groupings have reservation requirements that overlap (i.e. the same resource at the same timeslot), they will not be able to map to the same resource (i.e. to which the overlapping reservation pertains) to without violating dependencies. Based on these constraints, graph coloring can possibly resolve a lot of the challenges faced during the mapping process.

9.3 Resource reduction stage

During the mapping process, resources are allocated based on the mapping state of instructions. Some input port reservations are canceled without allocating an input port. They are canceled

(instead of used) when the dependency is mapped to an existing connection, rather than allocating a new one. However, while the input port was reserved, it might have caused instructions to map to other (perhaps newly allocated) FUs. This is needed to guarantee functional correctness of the generated implementations.

The resource reduction stage would initiate after the mapping process completes. It removes a FU and all related connections, and attempts to remap all instructions that were mapped to it. If it can do so without reallocating the FU, the configuration is reduced. This sometimes requires multiple FUs to be removed simultaneously before progress can be made.

Referring back to the generation method discussed in subsection 2.3.5, this resource reduction stage would be the refinement stage as applied by [15].

Bibliography

[1] TU/e Electronic Systems, TU/e Signal Processing Systems, Donders Institute, Radboud UMC, *Wearable Brainwave Processing Platform*, http://brain-wave.nl/ 5

[2] Kunjan Patel, Séamas McGettrick, Chris J. Bleakley, *SYSCORE: A Coarse Grained Reconfigurable Array Architecture for Low Energy Biosignal Processing*, IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines, May 2011. 5

[3] Kanishkan Vadivel, Mark Wijtvliet, Roel Jordans, Henk Corporaal, *Loop overhead reduction techniques for coarse grained reconfigurable architectures*, Euromicro Conference on Digital System Design, 2017. 12, 15

[4] Luc Waeijen, Mark Wijtvliet, *Coarse Grain Reconfigurable Architecture*, http://cgra.nl/ 7

[5] Mark Wijtvliet, Luc Waeijen, Henk Corporaal, *Coarse Grained Reconfigurable Architectures in the Past 25 Years: Overview and Classification*, International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS), July 2016. 7

[6] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, Mark Horowitz, *Understanding Sources of Inefficiency in General-Purpose Chips*, ISCA '10 Proceedings of the 37th annual international symposium on Computer architecture, Pages 37-47, 2010. 12, 14

[7] Michaël Adriaansen, *Code generation for a Coarse-Grained Reconfigurable Architecture*, Technische Universiteit Eindhoven, 2016. 11

[8] G. J. Chaitin, *Register Allocation and Spilling via Graph Coloring*, 20 Years of the ACM/SIGPLAN Conference on Programming Language Design and Implementation (1979-1999): A Selection, 2003. Volume 39 Issue 4, April 2004, Pages 66-74, Original: 1982. 13

[9] Akira Koseki, Hideaki Komatsu, Toshio Nakatani, *Preference-Directed Graph Coloring*, PLDI '02 Proceedings of the ACM SIGPLAN 2002 conference on Programming language design and implementation, Pages 33-44, June 17-19, 2002, Berlin, Germany. 13

[10] Pierre G. Paulin, John P. Knight, *Force-Directed Scheduling for the Behavioral Synthesis of ASIC's*, IEEE Transactions on Computer-Aided Design, Vol. 8, June 1989. 51

[11] A. Tiemersma, *Optimal Instruction Scheduling and Register Allocation for Coarse-Grained Reconfigurable Architectures*, Graduation Report, Eindhoven University of Technology, March 2017. 12, 17

[12] J. Hoogerbrugge, *Code Generation for Transport Triggered Architectures (par. 7.1, 136-141*, Dissertation, Eindhoven University of Technology, 5 February 1996. 15

[13] G. Dimitroulakos, M. D. Galanis, and C. E. Goutis, *Exploring the design space of an optimized compiler approach for mesh-like coarse-grained reconfigurable architectures*, in Proc. IPDPS, 2006, pp. 1–10. 20, 41

[14] Dongkwan Suh, Kiseok Kwon, Sukjin Kim, Soojung Ryu, Jeongwook Kim, *Design Space Exploration and Implementation of a High Performance and Low Area Coarse Grained Reconfigurable Processor*, 2012 International Conference on Field-Programmable Technology, Dec. 2012. 15

[15] R. Jordans, *Instruction-set Architecture Synthesis for VLIW Processors*, Dissertation, Technische Universiteit Eindhoven, 2015. 17, 81

[16] L. Józwiak, D. Gawlowski, A. Slusarczyk, A. Chojnacki, *Static Power Reduction in nano CMOS Circuits Through an Adequate Circuit Synthesis*, 14th International Conference on Mixed Design of Integrated Circuits and Systems, June 2007. 16

[17] Bingfeng Mei, Andy Lambrechts, *Architecture Exploration for a Reconfigurable Architecture Template*, 14th International Conference on Mixed Design of Integrated Circuits and Systems, June 2007. 12

[18] Dongrui She, Yifan He, Bart Mesman, Henk Corporaal, *Scheduling for register file energy minimization in explicit datapath architectures*, DATE '12 Proceedings of the Conference on Design, Automation and Test in Europe, Pages 388-393, 2012. 42

[19] Andy Lambrechts, Praveen Raghavan, Murali Jayapala, *Energy-Aware Interconnect-Exploration of Coarse Grained Reconfigurable Processors*, Proc. of Workshop on Application Specific Processors, 2005. 23

[20] Miguel R. Corazao, Marwan A. Khalaf, *Performance Optimization Using Template Mapping for Datapath-Intensive High-Level Synthesis*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, August 1996. 16

[21] Allan Carroll, Stephen Friedman, Brian Van Essen, Aaron Wood, Benjamin Ylvisaker, Carl Ebeling, Scott Hauck, *Designing a Coarse-grained Reconfigurable Architecture for Power Efficiency*, Department of Energy NA-22 University Information Technical Interchange Review Meeting, 2007. 20

[22] S. Jafri, M. A. Tajammul, A. Hemani, K. Paul, J. Plosila, and H. Tenhunen, *Energy-Aware-Task-Parallelism for Efficient Dynamic Voltage, and Frequency Scaling, in CGRAs*, in Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on, 2013, pp. 104–112. 21