TECHNICAL UNIVERSITY OF CLUJ-NAPOCA
FACULTY OF AUTOMATION AND COMPUTER SCIENCE

Internship report

CODE GENERATION FOR HIGH-LEVEL SYNTHESIS TOOLS

Student: Novacean Nicoleta Ligia
Group: 30434
Institution: Laboratoire de l'Informatique du Parallelisme(LIP)
École Normale Supérieure de Lyon
Supervisors: Matthieu Moy, Christophe Alias
Period: 09.07.2018 - 14.09.2018

Contents

1 Abstract

This internship took place at LIP (Laboratoire de l'Informatique du Parallelisme), a computer science laboratory in Lyon, associated with CNRS, ENS Lyon, INRIA and UCB Lyon 1. As an intern, I was part of a research team, namely CASH (Compilation and Analysis, Software and Hardware). The main objective of CASH is to provide energy-efficient software and hardware compilation techniques by taking advantage of architectures like multi-core processors, GPUs or FPGAs.

Today's FPGAs are an efficient solution for performing advanced computing. FPGA accelerators allow for tremendous parallelism: one board provides parallel computing resources numbering in millions. Usually, FPGAs are configured by means of Hardware Descriptive Languages (e.g. Verilog, VHDL). However, for reducing the development effort and the need of FPGA electronics knowledge, High-Level Synthesis languages like Vivado HLS have been introduced. Vivado HLS takes as input a C specification and generates the corresponding HDL description. Vivado HLS can structure functions in the input code as a single state machine or extract a limited level of parallelism from it.

One of CASH team's projects has as objective mapping C code to massively parallel FPGA accelerators. For enhancing the level of extracted parallelism, the team developed Dcc-light. Dcc-light is a parallelism extraction compiler that outputs a dataflow representation named DPN (data-aware process network). Therefore, the input C code will not be mapped directly to an FPGA board using Vivado HLS. Instead, parallelism is extracted from the input program using Dcc-light as a DPN program. Afterwards, we convert the DPN intermediate representation into a subset of C code. The obtained C program is provided as input to Vivado HLS, which synthesizes the circuit to be mapped on the FPGA, as seen in Figure 1. Besides these transformations, the DPN is also used within SystemC code generation for simulation purposes.

The main tasks I was in charge of were to generate and optimize the C code for high-level synthesis. My work included experimenting with several tools and topics such as Vivado HLS and its optimization strategies, the DPN intermediate dataflow representation and, finally, a code generation algorithm.

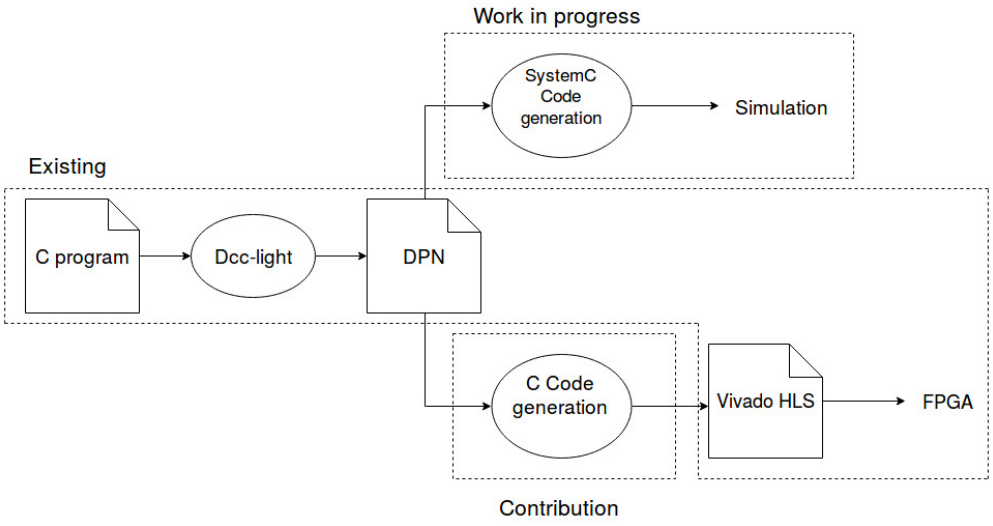

Figure 1: Project Diagram

2 Introduction

2.1 Problem Statement

Nowadays, the need for high-performance computing is increasing. This results in an increased level of parallelism, hence the need for high-level code optimization and improved compilers.

Prior to 2006, given the decrease in transistors' size, circuits could operate at a higher frequency for the same level of power consumption. However, now it is not longer possible to keep the typical power consumption constant as new generations of processors appear. Issues like power efficiency, data movements and memory imposed bounds need to be tackled in an innovative manner. Computer systems need to be reorganized at all levels: hardware, software, compilers and runtimes.

The vision of CASH team is that parallelism based on dataflow is a good model to handle HPC applications. The dataflow representation illustrates data movement as it passes from one component to the next, taking into consideration how computations within the process change the data.

The DPN model, standing for data-aware process network, is the intermediate dataflow representation of the Dcc parallelism extraction compiler developed by the team. This parallel execution model is composed of processes that communicate with each other through buffers. It provides the advantage of expressing both the data and task level parallelism. Also, it can used as an intermediate language in HPC computing as compilation is possible both to and from dataflow program representations.

Also, the data-aware process network is adapted to the hardware constraints of high-level synthesis. As previously mentioned, the final step of the project is mapping onto an FPGA board. Given the massive parallelism FPGAs offer, functions' execution only requires a few clock cycles. This allows for a smaller clock frequency, thus lower power consumption, while still obtaining a boost in performance. One way to configure the FPGA is through VHDL code, but development using hardware description languages is not one of the team's goals. Therefore, one way of obtaining the circuit design is by means of high-level languages and hardware compilers that enable C or C-like code to be synthesized into an RTL description. An example of such a high-level synthesis tools is Vivado HLS.

2.2 Research Contribution

The overall objective of the internship is to transform and schedule the DPN dataflow model to meet optimization criteria such as throughput, I/O traffic or memory requirements. More precisely, the work includes:

1. Study of the DPN and Vivado HLS tools by implementing in C and optimizing of a set of examples. The examples include: computing the pair-wise sum of two vectors, the Jacobi method in 1D or the Cholesky decomposition.

2. A code generation algorithm that converts the data-aware process network representation into a subset of C code. This transformation must be done according to the restrictions imposed by Vivado HLS.

3. Optimization of the above item using the strategies offered by the Vivado HLS Tool. Finally, Vivado HLS produces the circuit description to be mapped onto the FPGA board.

3 Technical Description

3.1 Background

3.1.1 Polyhedral Model

The polyhedral model is a mathematical framework that offers a powerful abstract representation for reasoning about nested loops optimization, parallelization or scheduling. Each iteration of a statement is viewed as a point inside the polyhedron defined by a set of affine constraints. These affine constraints are inferred from the bounds of the surrounding loops. The reunion of all these points, so all the values of the surrounding counters for which statements will be executed, forms the iteration space.

Let us consider the following C example:

```
const int T = 4;
const int N = 5;
int a[N];
int i,t;
for (i = 1; i <N; i++)
    for (t = 1; t <T; t++)
        a[i] = a[i]/2;
```

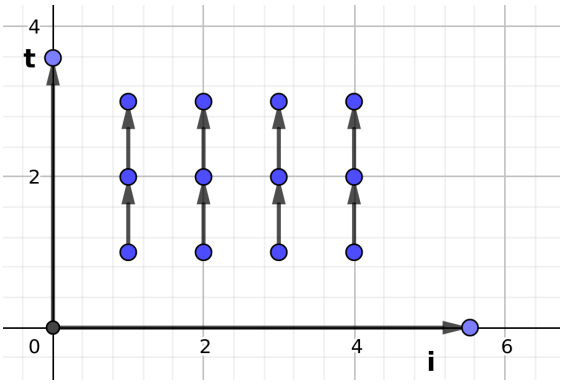

Figure 2: Iteration space and dependencies between statement iterations

The representation of the iteration space and of the dependencies between statement iterations for the previous code can be found in Figure 2.

The dependency between two points is characterized by the dependency depth. The dependency depth is the level of indentation at which the dependency is solved. In our example, the dependence is:

$$D: (i, t) \rightarrow (i, t+1),$$

where (i, t) is the source of the dependency and (i, t+1) the target.

The scheduling function θ gives the execution date. By applying θ to the source and function, we get:

$$\theta(i,\ t) = (i,\ t)$$
$$\theta(i,\ t{+}1) = (i,\ t{+}1)$$

Now, the obtained tuplets will be lexicographically compared. The dependency depth is the index where the comparison may stop, starting from 0. Therefore, for our example the dependency is solved at the level of t and the dependency depth is 1.

One of the loop transformations for optimizing nests that the polyhedral model supports is loop tiling. Loop tiling separates the iteration space into tiles (smaller groups of iterations) such that the data accesses inside the same tile make efficient use of one or more levels of cache, while favorizing reuse across tiles. After applying loop tiling on the example, the way a loops iterates over its domain will change. Therefore, vertical tiles are obtained and the code from different tiles can be executed in parallel, while the code for each tile is executed sequentially.

3.1.2 DPN

DPN (Data-aware Process Network) is a dataflow intermediate representation used for extracting parallelism from the C code provided as input. The process of parallelism extraction relies on the polyhedral model framework. The resulted compiler is capable of producing a DPN representation without deadlocks for polyhedral programs.

The DPN processes can be of 3 types: Load, Store and Compute. Communication from one process to the next is done through buffers. Besides arrays and single-valued, these buffers may also be FIFOs with only one producer and one consumer. As it has been noticed during the internship, FIFO buffers play a major role in generating an optimized parallel circuit design.

Inside DPN, each statement of the program is associated to a compute process. As inspired by the polyhedral model, processes are defined by:

1. Iteration vector: column vector where each entry corresponds to a surrounding loop index
2. Iteration domain: set of values of the iteration vector for which the process is executed
3. Indexing function: mapping for the points inside the iteration domain. The new points are the coordinates of the next process instance

3.1.3 Vivado HLS

- Vivado HLS Projects

A diagram containing the steps and components of a Vivado HLS project is illustrated in Figure 3.

The primary input of a Vivado HLS project is the C function to be synthesized, labeled with 1 in Figure 3. The C program can be written in various languages, including C, C++ and SystemC. An important remark is that the code of the function to be synthesized must follow some restrictions imposed by Vivado HLS:

1. System calls to the operating system: The entire functionality of the function must be specified without using any system calls to the OS. As a remark, Vivado HLS ignores system calls that do not have any effect on the execution of the program (e.g. fprintf(stdout)).
2. Dynamic memory allocation: The used data structures must have fixed size. The resources on an FPGA board are limited and fixed and dynamic allocation or deallocation of memory resources will not be synthesized. For a design to be synthesized to a RTL description, it must be fully self-contained, clearly indicating all the needed resources.

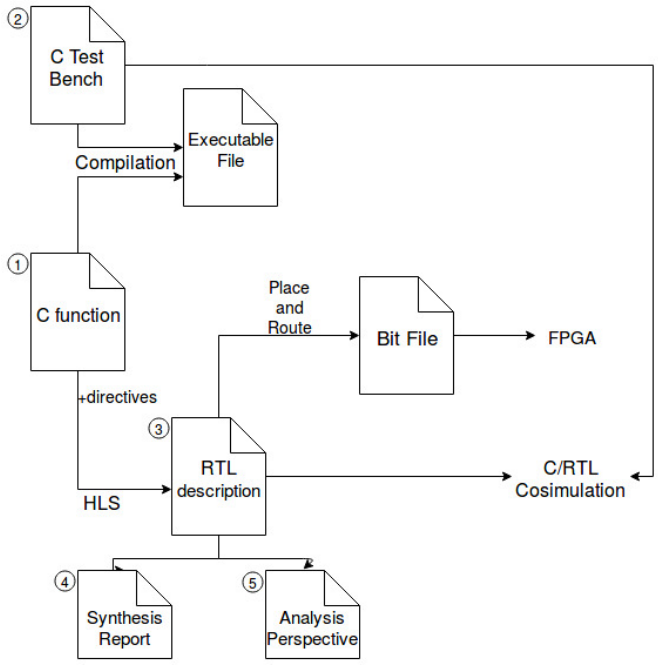

Figure 3: Iteration space and dependencies between statement iterations

3. Recursive functions: Recursive functions that may generate endless recursion will not be synthesized.

Labeled in Figure 3 with 2, the C test bench has an important role in the flow of a Vivado HLS project. Firstly, the test bench is used for checking the correctness of the top-level C function before synthesis. As RTL simulations execute much slower than C programs, it is more efficient to test the algorithm before we proceed to the synthesis step. Secondly, the test bench generates the input vectors for the RTL design and verifies the RTL output at C/RTL co-simulation time. The test bench should check the results outputted by the top-level function. This way, the outputs produced by the RTL description of the top-level function can be checked by simulation.

The RTL (register-transfer level) description format, corresponding to label 3 in figure 3, is an abstraction used to represent the way data changes as it flows from one register to another. Data transformations are performed by combinational circuits placed between registers. Hardware Description Languages (HDL) rely on RTL for creating a high-level representation of a circuit, without describing the actual hardware that will carry out the operations. The RTL design is, further on, turned into a lower-level representation and mapped to a circuit.

The Synthesis Report, labeled with 4 in Figure 3, shows performance metrics like:

1. Area: used amount of FPGA hardware resources (look-up tables, registers, block Rams, DSP48s).

2. Latency: number of clock cycles that a function need for outputting all the results

3. Initiation interval: number of clock cycles before a function can accept new input data.

4. Loop iteration latency: number of clock cycles that a loop iteration for executing.

5. Loop initiation interval: number of clock cycles before the next loop iteration can accept new data.

6. Loop latency: number of clock cycles required for executing all the iterations of a loop.

The Analysis perspective is a Vivado HLS feature used to analyze the design obtained after HLS, as shown by label 5 in Figure 3. This perspective consists of several panes, one of them being, in Vivado HLS 2018, the Schedule Viewer. The Schedule Viewer shows the performance details for the selected level of hierarchy. A remark is that it does not illustrate the way operations are scheduled every clock cycle, but in control steps. A control step is an internal state that HLS uses to schedule operations. Also, there exists a correlation between control steps and the states in the RTL Finite State Machine, but it is not a one-to-one mapping.

- Simulation, Synthesis and Co-simulation

A Vivado HLS project consists of 3 major steps: C Validation/C Simulation, High-Level Synthesis and RTL Verification.

1. C simulation is the name used in HLS for the execution of a compiled C program. This step has the purpose of verifying the correctness of algorithm, by using both the C function and the C test bench. The possibly applied optimization directives do not have any effect on the result of the simulation. As a remark, the C test bench and the file containing the C function may also be executed outside the Vivado HLS context, as illustrated by Figure 3.

2. High-Level Synthesis is when the C design is synthesized into an RTL (Register-transfer Level) design. When optimizing the design, improperly applied directives result in errors or warnings at synthesis time. However, a successful synthesis does not necessarily imply a correct design because there are issues detected only in the last step (e.g. a deadlock situation on the dataflow memory channels).

3. The last step is RTL Verification. Due to using both the C test bench and RTL output from High-Level Synthesis, the step is often called C/RTL co-simulation. The generated simulation report indicates if the simulation succeeded and the measured latency and interval. As noticed, these values are not the same as the ones obtained after synthesis. Finally, we may have access to waveform trace data, which allows us to view and analyze the RTL waveforms and gain insight into the way our optimized module works.

- Optimization techniques

High-level synthesis automatically tries to minimize latency of loops and functions, so it tries to execute as many operations as possible in parallel. However, for having the guarantee of obtaining the needed performance and transforming our implementation into a high-throughput and low-area implementation, we should use optimization directives. These directives are used to pipeline tasks, modify the way arrays (block RAMs), loops, functions or ports are implemented or providing information about the presence or absence of data dependencies, all these enabling the usage of more optimizations.

Two of the most powerful optimization directives are PIPELINE and DATAFLOW. They are of great use when we try to obtain a high throughput and small initiation interval.

1. The PIPELINE directive

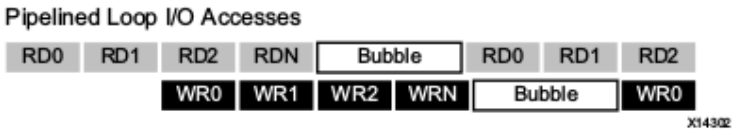

Figure 4: Pipelined loop I/O accesses (taken from *Vivado Design Suite User Guide. High-Level Synthesis*)

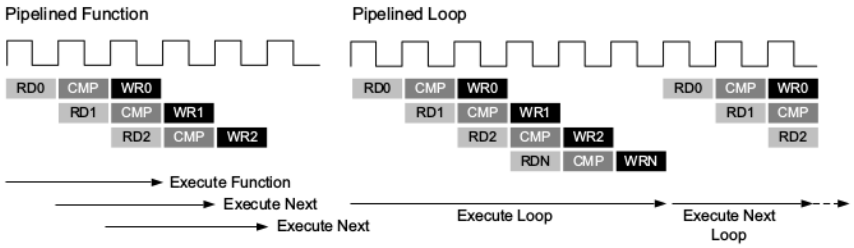

Figure 5: Pipelined function and loop (taken from *Vivado Design Suite User Guide. High-Level Synthesis*)

It reduces the initiation interval of functions or loops by allowing operations to execute in parallel. A statement execution does not need to have all its operations completed before it can start a new operation.

The behaviour of pipelined loops and pipelined functions is different. The function pipeline runs forever and accepts new inputs continuously. On the other side, the loop pipeline executes until all the iterations of the loop are completed. Before being able to start the next loop, a loop must finish all its operations. Figure 5 illustrates this behaviour. When pipelining loops, a so called "bubble" is generated, as seen in Figure 4. Loops used in a dataflow region or as top-level loops in a function can reach continuous execution using the rewind option when applying the PIPELINE directive.

The initiation interval is a metric with great importance when pipelining loops. It is important to keep in mind the following fact: even if the loops contained in a function reach the point where they process a sample every clock cycle, the initiation interval of the function is reported as the time it takes for the loops inside the function the finish processing all the data.

2. The DATAFLOW directive

   The dataflow optimization represents a "coarse grain" pipelining at function and loop level. It is highly used in improving the throughput of the design. When Vivado HLs identifies a dataflow region, it creates channels for storing the result of each task. These channels are implemented either as FIFOs or ping-pong buffers. The entire functionality of the dataflow relies on HLS being able to tell when data from the first function will be available for the second function.

   Even if it represents a very powerful method of performance improvement, there exist a series of restrictions regarding the coding style it supports. Respecting the following coding styles is recommended in order to be sure that the dataflow optimization will be applied.

- Single-producer-consumer: All the data flowing between tasks must be driven by only one task and consumed by only one task.
- No bypassing tasks: Data must flow from one task to the one right after it, in the sequential execution order.
- Feedback between tasks: Inside a dataflow region, data must only flow in a forward direction. Then, the output from a task cannot be consumed by a previous task.
- Conditional execution of tasks: Data must flow from one task to the next one. Instead, conditional execution may result in skipping one task. One way to avoid this problem is by moving the conditions inside the function body.
- Loops with multiple exit conditions are not allowed.

3.2 Contribution

3.2.1 DPN and Vivado HLS optimization on examples

As previously stated, the code generation algorithm outputs the C specification of the top-level function, together with all its subfunctions, that will be synthesized by Vivado HLS. One of the examples used for studying the DPN to C conversion is the Jacobi method in 1D.

Firstly, we will provide the C code of Jacobi 1D, which is the input of the entire chain of transformations.

```
a[0] = 0;
a[N-1] = 0;
for(t=0; t <= T-1; t++)
    for(i=1; i <= N-2; i++)
        b[i] = (a[i-1] + a[i] + a[i+1])/3;
    for(i=1; i <=N-2; i++)
        a[i] = b[i];
result = a[1];
```

The DPN representation will contain 5 compute processes, one for each assignment. By analyzing the previous code, we notice that the dependency between the assignments *b[i] = (a[i-1] + a[i] + a[i+1])/3* and *a[i] = b[i]* is a backward dependency: the first assignment reads data written by the second one. This dependence will result in a feedback between tasks in the DPN. Each DPN process contains the loops corresponding to the iteration domain of the assignment they are associated to. However, if we extract the loop with index t, which is common to both assignments, we eliminate the feedback between tasks.

According to the identified loop carried dependencies, the calls to functions implementing processes in the generated C code are organized the following way:

```
process_LOAD_0();
process_COMPUTE_1();
process_COMPUTE_2();
for (int t = 0; t <= 15; t++) {
    process_COMPUTE_3(t);
    process_COMPUTE_4(t);
}
process_COMPUTE_5();
```

process_STORE_6();
process_STORE_7();
process_STORE_8();

Now, optimization techniques should be applied to the obtained C program. First of all, the DATAFLOW directive will be used for enabling task-level parallelism. Functions and loops will be allowed to overlap their execution, so the throughput of the design will be improved. Then, we will also apply a "fine grain" pipelining at the level of operators. This is done by the PIPELINE directive.

3.2.2 Code generation

The C code generation algorithm has as input the DPN program and outputs a C specification. The DPN representation is accessed via an interface that provides information about all the components and their connections (processes, buffers, synchronization, multiplexers and demultiplexers used in process-buffer connections). The shape of the C output program is influenced by the organization in processes that characterizes DPN, but also by the restrictions imposed by Vivado HLS as mentioned in section 3.1.3.

Vivado HLS imposes the existence of a top-level function that will be the object of HLS. Its arguments will be synthesized as I/O ports and the functions it calls will become blocks in the RTL hierarchy. Particularly, arrays are by default synthesized as block RAMs or, if arguments of the top-level function, as ports for accessing block RAMs outside the internal design.

The first version of the code generation algorithm is a limited version that can only be used for cycle-free DPN representations. Let us consider the following example:

f(buffer1, buffer2); // buffer1 is read, buffer2 is written
g(buffer2); // buffer2 is read

An important note is that functions f and g contain the loops corresponding to the iteration domain of the statement they implement. The previous code contains no backward dependency between processes. Applying the dataflow directive is possible in this case. However, let us consider one more example:

f(buffer1, buffer2); // buffer1 is read, buffer2 is written
g(buffer2); // buffer2 is read
h(buffer1); // buffer1 is written

The previous example contains a backward dependency between functions f and h, as h writes buffer1, while f reads it. Due to the feedback between tasks, the dataflow directive fails to be applied.

Therefore, a second version of the algorithm solves the issue of feedback between tasks. Loops carrying dependencies from inside the function are now extracted outside the function. The algorithm has three major parts: cycles detection and loop extraction code generation, generation of functions for each process and generation of the top-level function.

- Cycles detection and loop extraction code generation

  As previously mentioned, DPN processes may communicate via non-FIFO or FIFO buffers. The functionality of the DPN relies on blocking reads and writes. For example, if one process tries to read data that is not available, it will wait and be notified when data

is available for reading. This way, processes that execute inside a cycle can exchange data while being executed in parallel. The issue is that, at simulation time, processes are executed sequentially. This means that a process will not read data as soon as it is available, but only when the previous process finished all its operations. This improper data exchange, critical for the DPN representation, results in wrong results, so a failed simulation. Furthermore, the DPN structure contains feedbacks between processes that are not supported by the DATAFLOW directive, which has an important role in increasing the throughput of the design.

The solution that we have found for both issues is extraction of loops carrying dependencies. Loops carrying dependencies are materialized in the DPN structure as cycles, so we must detect cycles inside DPN. Then, we detect the loop indices that are common to all the processes inside the cycle and those will indicate which loops need to be extracted. By extracting the common loops, data will be passed from one process to the next one at every iteration, as specified in the initial C code. A particular case is represented by loops that are common to multiple cycles. For example, if one process belongs to more cycles, there exists a posibility of needing to create nested loops containing the processes of more cycles in order to solved the feedback between processes.

- Function generation for each process

  Each DPN process is translated into a function that handles the input selection (done through multiplexers), output computation and output writing to buffers. The function name includes the type of the process (load, store or compute) and the rank, which is the unique identifier inside DPN. The following components need to be generated:

  1. Parameters

     The parameters of the functions are: indices of the loops that carry dependencies, the variable to load (for LOAD processes), the variable to store data to (for STORE processes) and all the buffers used within the process.
  2. Loop nests

     The DPN interface provides a function which prints the loop nests corresponding to each process based on the provided surrounding counters. Therefore, for a process which is not part of a cyclic dependency we provide the surrounding counters as they are in stored in the DPN program. However, if the process is part of a cycle, we need to update the surrounding counters by removing the ones that carry dependencies.
  3. Multiplexers

     Each MUX provides the process with one of the values it needs for executing the statement. Based on a set of constraints on loop indices, only one of the inputs of the multiplexers will be selected. The sets of constraints are disjoint.
  4. Statement

     Each assignment from the C code provided as input to Dcc is associated to a compute process. The statement is the representation of the assignment from the initial C code in terms of data coming from communicating buffers. Each process will compute only one output based on the inputs provided by the multiplexers.
  5. Demultiplexer

     The demultiplexer selects the buffers where the previously computed value should be stored. The clauses of the DEMUX are not disjoint, so the same value may be written into several buffers. A particular case appears for LOAD processes, where the stored value will be an item from the data structure to load.

- Top-level function generation

  The arguments of the top-level function will be the I/O ports of the design. They must be specified by the user in an XML file as a *map<string, string>*. The key should be the name of the variable and the value is its declaration.

  The body of the top-level function starts with the declaration of the buffers used by DPN. A buffer is uniquely identified by a rank. The type of the buffer (FIFO, non-FIFO) is also available. In case of FIFO buffers, they are declared using

  $$static\ hls::stream\ <datatype>$$

  The calls to the functions implementing processes are done in a sequential manner until a process that is part of a cycle is met. Then, the loops carrying dependencies are produced. Inside the generated loop nest, we perform calls to the functions implementing processes that are part of the cycle, in the order imposed by their ranks. Cycles are not necessarily formed by processes with consecutive ranks. Therefore, the detected intermediate processes are generated before the loop nest.

4 Conclusions and Further Work

4.1 Technical Conclusion and Further Work

The work done on finding the proper structure and optimization strategy of the intermediate C code offered a better understanding of the restrictions and abilities of Vivado HLS. Current knowledge suggests that the restrictions imposed by the usage of the dataflow optimization, along with the requirement of a successful simulation, may raise issues in optimizing the design for the highest throughput. The issue of loop carried dependencies should be studied into more depth, as well as the behaviour of designs under the effect of combined optimization directives.

A better loop extraction mechanism may be achieved by replacing the extraction of all the common loops with the extraction of the loop carrying the dependency with the smallest depth. Once the optimization restrictions are met in an as effective as possible manner, the design may be mapped onto FPGA boards for a final validation of the results.

4.2 Personal Conclusion

This internship has been a great opportunity for me -an undergraduate student who tries to figure out her career- to gain insight into the research field. As intern in a research team, I have witnessed both the satisfaction of solving an issue, but also the disappointment of temporarily facing a problem for which you cannot find a favorable solution. Probably the most helpful lesson I've grasped is that admitting your approach may not be the adequate one for the given context is not always a step back, but it might be a step forward for a better final result.

It was an honour to be surrounded by hard-working, intelligent and, most importantly, dedicated people. I have understood even more than before the importance of doing your job with pleasure, because along the way lots of difficulties appear and a strong enough motivation may only be fed by true dedication. I believe this is even stronger in research. Working with a small number of partners seems demanding and maximum commitment and responsibility are necessary for reaching your goals.

On the technical side, I have acquired knowledge in hardware related topics, C and C++ programming languages and also learnt how to properly write a scientific document. All these will certainly be of great help in my future academic and work experience.

I am grateful to the people from LIP for welcoming me, especially to team CASH for all the help and support they have provided me with.