Technical University of Cluj-Napoca
Faculty of Automation and Computer Science

internship report

Code Generation for Simulation using SystemC

Student: Alexandra Dobre
Group: 30431
Institut: École Normale Supérieure de Lyon
INRIA - Institut National de Recherche en Informatique et en Automatique
Supervised by Matthieu Moy
Christophe Alias
Period: 7th of July 2018 - 14th of September 2018

2018
September

MINISTRY OF EDUCATION

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA, ROMANIA

Contents

1 Abstract

The internship took place in Lyon, France, at **LIP** (Laboratoire de l'Informatique du Parallélisme), which is associated with **CNRS** (Centre national de la recherche scientifique), **ENS** (École Normale Supérieure) de Lyon, **INRIA** (Institut National de Recherche en Informatique et en Automatique) and **Université Claude Bernard Lyon 1**. The team in which I worked is the **CASH team**. Its name stands for Compilation and Analyses, Software and Hardware, which slightly sums up their interest and vision.

The need of optimized software or hardware compilation for high-performance computing (HPC) with data-intensive computations has been noticed during the last years. It is the goal of the CASH team to find ways in which complex HPC applications can be handled. The transverse and fundamental topic of CASH can be expressed in one word: **dataflow**. The CASH team wants to create a flow, expressed in Figure 1, in which C code can be processed into parallel code using a suitable dataflow model. After that, the code has to be synthesized using high-level synthesis tools. In the particular case of this research, the chosen tool is Vivado HLS. The synthesized code will be finally mapped on the FPGA. Simulation is a very important part of complex hardware programming, since it provides an environment in which debugging takes less time, no setup time is needed and it helps to discover whether the problem appears in hardware or in software. In order to fulfill the task efficiently in terms of time and resources, simulation of the generated code is highly needed. Another reason why simulation is required is the fact that the dataflow model is hard to debug and the code used for simulation would help to easily detect bugs.

The task that I worked on was writing a tool which provides **code generation for simulation in SystemC**.

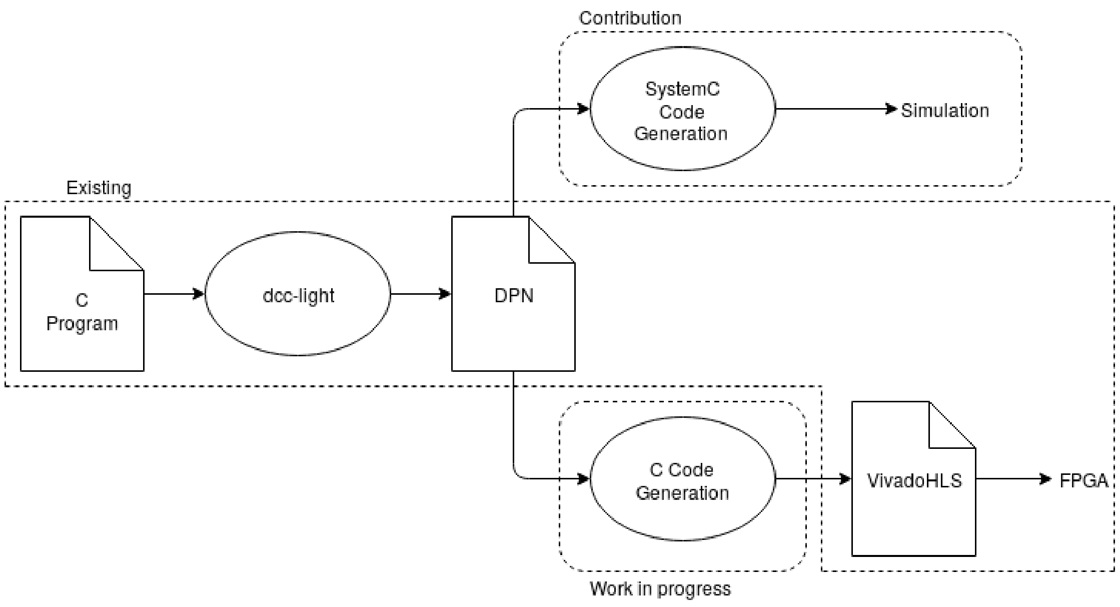


Figure 1: Project flow

2 Introduction

The increased level of parallelism in supercomputers, but also in classical end-user computers, has led to the need for better compilers and high-level code optimization. Not long ago, the evolution of every computing system was described by the Dennard scaling. This is a fact enunciated by a researcher with the same name, saying that "as transistors are reduced in size, their power density stays constant" [1]. Since Dennard scaling no longer applies, one of the tasks is to find new ways of improvement in computing systems at all levels: hardware, software, compilers and runtime. On the hardware side, several new architectures are introduced: multi-core processors, Graphics Processing Units, many-core and FPGA accelerators. A consequence of this heterogeneity is the diversity of ways in which a given computation can be implemented. High-level languages and hardware compilers are required.

Parallelism based on dataflow is one approach to handle these above-mentioned issues. The dataflow program consists of a set of processes that communicate through buffers. The strategy used in this research was based on the DPN (data-aware process network), a dataflow intermediate representation for parallelizing compilers. Another interesting topic discussed was the improvement that the polyhedral model can bring to the parallel implementation.

The CASH team focuses, among other related topics, on extracting the parallelism in the form of a DPN and producing efficient code for hardware accelerators such as FPGAs. The flow is described in Figure 1. The team has developed a compiler for C-code which extracts the parallelism and expresses it with the help of a dataflow model named DPN. Simulation of the intermediate representations, which are expressed as process networks, has to be done without actually using the FPGAs. Debugging dcc-light directly on FPGAs would be very costly and very limited. On the other hand, passing through all the steps of the synthesis would take too long. The solution currently used is the POSIX-thread based simulator which uses one thread for each network process. Despite all the benefits it brought, the POSIX thread simulator is harder to use in cases of debugging because it can not handle the frequency of context changes [2].

The proposed solution is to use a simulator dedicated to hardware simulation: SystemC. SystemC is a C++ library which provides event-driven simulation code. It mimics the hardware description languages, but is more of a system-level modeling language. The overall goal of the internship is to find the benefits and the drawbacks of a SystemC-based simulator. Also, a code generator for this SystemC simulator has to be implemented. The code generator needs to take the parallel DPN representation of the initial C-code and turn it into SystemC.

3 Technical Background

In order to start working on the task proposed, one has to become familiar with some technical information. This technical issues are either used in the previous work of the team, or are part of the development needed for the task proposed.

3.1 Polyhedral models

In the context of a growing need to optimize the programs and to parallelize them, the programs have to be understood and transformed into faster ones or into parallel programs. This complete transformation can not be done with unbounded loops. The difficulty in case of

[1] *Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions* - Robert H. DENNARD, Fritz H. GAENSSLEN, Hwa-Nieu YU, V. Leo RIDEOUT, Ernest BASSOUS and Andre R. LEBLANC, April 1999

[2] See `http://www.ens-lyon.fr/LIP/CASH/`

unbounded loops arises because the number of points in the iteration domain is not known. A solution for programs that need to be improved, but encounter the issue of unbounded loops, is to use the affine control loops. Compared to the unbounded loops, the affine control loops can be represented and transformed symbolically. The polyhedral models are useful because of the geometric operations and linear optimization provided. Polyhedra are used while working with loop kernels and complex data structures, because iterations in affine loops can be represented as points in a polyhedron. During the research previously done by the team, it has been shown that programs can be processed easily to a dataflow model, namely the data-aware process network, using polyhedral techniques.

The polyhedral model is used when computations that needed to be done are too large.

3.2 Data-aware Process Network - DPN

Dataflow models are one of intermediate representations available for parallelizing compilers. Data-aware process network (DPN) is a parallel execution model which works by means of extracting the parallelism from the regular computations. After all the previous research done on dataflow models, it has been shown that DPN is a good way to provide the support of the code generators. One of the reasons why DPN is used in this research is that it mainly focuses on data movements, which are an important issue in parallel computation.

A dataflow model usually divides the computation into processes and creates a way of communication between them by using buffers. The DPN consists in 3 possible processes: load, compute and store. The load process is the one in which data is taken from the input and loaded on the buffers. The compute process is the one in which computation takes place. There can be more than one compute process in a program, depending on the complexity of the code. In the store process, data is taken from some buffers and stored into others or given as an output. Data is moved from one process to another in parallel with the help of buffers. Buffers play the role of connecting processes one to another and of transmitting the data between processes. They can be either single-valued, arrays or special buffers of type first-in, first-out (FIFO).

3.3 SystemC

Recently, researches had to deal with an increase in the complexity of hardware and software due to the evolving high-performance computing. The traditional simulation techniques have been influenced by this increase and they have become too inefficient and slow. There are new techniques which can be followed in order to get improved results and the one highly used by industry is Transaction Level Modeling (TLM) in SystemC. This increasing complexity motivates the choice of using SystemC in this research.

SystemC is a C++ library which provides a simulation interface with many levels of abstraction, integrating both hardware and software. SystemC is an open source library often chosen because of the fast simulation which anyone - industry, universities or research institutes - can use. This library can be compared to VHDL or Verilog in its organization and a bit in the coding style, but the syntax is C++. SystemC has two types of libraries: the master/slave-type library and layered libraries such as TLM or verification libraries.

Some important features of the SystemC library are:

- **Modules.** The container classes in SystemC are called modules. A SystemC module can have many processes that can communicate via ports.
- **Ports.** Communication between two or many modules via channels is possible by means of ports.

- **Signals.** In SystemC, signals can be either resolved - having more than one driver, or unresolved - having only one driver.
- **Processes.** Processes in SystemC are written inside the modules. They are used to describing functionality and are concurrent. In SystemC, they are the main computation elements.
- **Channels.** Channels are the elements through which communication is made in SystemC. They can be signals, buffers, FIFOs, mutex or semaphores. The equivalent of wires is the signals.
- **Interfaces.** Interfaces are used by ports to communicate with channels.
- **Events.** In SystemC, events are used to perform synchronization between processes.

The data types available in SystemC are similar to the one used in hardware programming, such as bits and bit vectors, 4-valued logic, arbitrary precision integers or fixed-point types. SystemC uses some classes and data type which describe time, which are useful for simulation. Other C++ language standards are used during programming with SystemC.

Since parallelism is one of the main goals, but also one big challenge of the research, the usage of SystemC is motivated by the scheduler it contains. This scheduler is able to manage a list of processes and an event list.

4 Contribution

The flow described in Figure 1 can be partitioned in 3 branches. The first one is the work already done by the team, in which the C-code in its original form is processed into a parallel representation - the **DPN** - by the **dcc-light** compiler. The second branch, which is still in progress, describes the synthesis steps, which consist in taking the DPN representation and turning it into parallel C-code or C-like code. This newly generated C-code is passed through the **Vivado HLS** tool and synthesized on the FPGAs accelerators. The third part is the task mainly assigned to me, where the information from the DPN is processed through a code-generator into SystemC-code ready for simulation. This step is necessary for a better development and debugging of the DPN, but also for easily debugging before the execution on the FPGAs.

The research studies the execution of 3 C-code programs: the addition of vectors, Jacobi-1D and Choleski. The 3 programs are compiled with the help of the dcc-light compiler. The compiled result for each one is given as a parallel representation in the form of the DPN. The DPN representation must be converted into code used for simulation, represented with the help of SystemC. Firstly, the 3 examples are written in SystemC code manually, in order to get an approximation of the code which will further be the output of the code generator for simulation. This has to be done in a generic manner, since the code generator has to work on numerous examples. Also, implementing the algorithms in SystemC is a good way to get comfortable with the syntax and especially with the particularities of this programming language. The code generation process reveals a lot of things about the correlation between DPN and SystemC.

4.1 Observations over SystemC implementation

SystemC is an event-driven simulator. SystemC code is divided into modules. They are represented hierarchically, so usually a toplevel module is used as a for connecting the rest of the modules needed in a program. An implementation of a program can have one topmodule component and many other module components. An advantage in SystemC is that it reuses a lot of already used basic components. The same module component can be used in as many topmodule components as wanted. Module components can be composed by other module components, when necessary. For example, for adding 2 vectors, there are used one module for performing the load from memory in the buffers, one module for computing the addition and one module for storing the result in the buffer used as on output. This hierarchy is explained in Figure 2 Each module has a constructor in which the instances of that module are created and initialized. Each function in the constructor is therefore used in the execution of the module. The variables are mainly declared and initialized in the private part of each module.

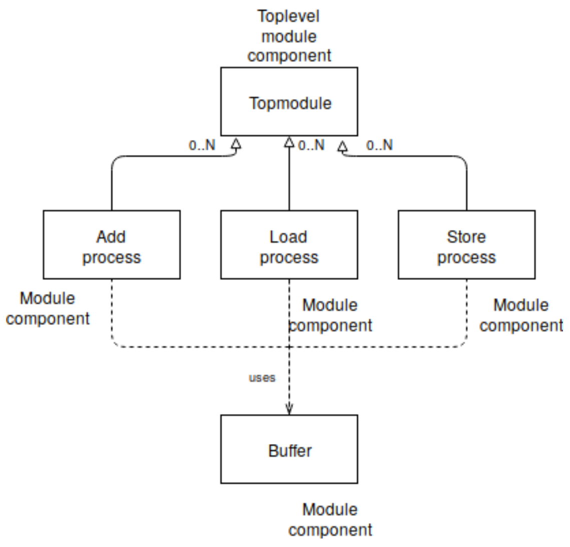


Figure 2: Adding two vectors in SystemC - hierarchy representation of modules

In SystemC, communication via channels is done with the use of ports. Channels can be signals, buffers, FIFOs, semaphores or mutex. The buffers needed in the implementation of the 3 examples have different dimensions. For example, if a buffer is single-dimensional, then it has to have the property that no matter how many times data is read from it, that data is not deleted. In case of the multi-dimensional buffers, then data will be written in the buffer and read from it multiple times, so it can use a regular read and write. This difference occurs mostly because some buffers are re-written, while some are not. A special module for the implementation of operations on buffers was written and used during the development of the entire task. For example, the module used for the operation with buffers implements a function which permits reading a value from a buffer without consuming it. Therefore, after the value is read from the buffer, it can still be found at the same location in the buffer, without being deleted. Channels of type FIFO are also used in the implementation of the 3 examples. For this, the sc_FIFO channel from SystemC is used. FIFOs in SystemC have a fixed length. Operations such as read and write are already implemented and can be used on SystemC FIFOs. During the implementation, an important thing about the buffers was noticed. In order to avoid deadlocks when the read from a buffer is done, the size of buffers is incremented by one.

Synchronization is a key element in the execution of SystemC code. In order to provide a safe concurrency implementation, SystemC provides the event-driven based mechanism. For example, when it is needed to store data into a buffer, an operation of read has to be performed first, followed by the operation of write. This synchronization is done by the use of sc_event. The event is notified at the occurrence of another event, making the process triggered or resumed. This can be expressed in the implementation of a store module in Jacobi-1D, when the input of a store process can be one of 3 initial choices. The decision of which one of the 3 choices will be stored is taken according to the point the iteration has reached. The decision procedure is expressed as a multiplexor. Next to the function in which the multiplexor is implemented, another function which takes the input and stores it in the output buffer is written. Since only one of the 3 choices will be stored, the function which writes the output has to wait for the decision to be made every time we move to another point in the iteration. This is illustrated in the code below:

```
 void read() {
   while (true) {
      // implementation of the multiplexor for the read operation
      // when the value is read from the buffer, the event is noticed
      m_e.notify(sc_core::SC_ZERO_TIME);
      // wait for the next iteration to be possible
      wait(m_next_iteration);
   }
}
void compute() {
   while (true) {
     // wait for the read to be finished
      wait(m_e);
      // implementation of the write operation
      // after writing the current value, the iteration can continue
      m_next_iteration.notify();
   }
   std::cout << "Done" << std::endl;
}
```

Templates are used for operating with generic types and generic functions. Therefore, an adaptive function template can be created. This allows to change the type or the function without repeating the entire code for each type or for each function. For example, some functions have type templates. Other functions, where increment by one is needed, have a template for both the type and the function.

4.2 Code generation

For writing the code generator, an interface of the DPN providing information about the parallel representation was used as a support of the implementation. Also, the characteristics of the DPN are very important in the implementation of the code generation, since a lot of the structures and the functions described for the DPN in its interface are used. The code generator outputs all the modules into the same file, but the hierarchy is kept the same as before. Each function in the code generator is written as factorized as it could have been, since the outline of each module is similar. The differences between processes are small and are mostly given by the DPN characteristics.

The beginning of each file where the implementation of modules is written has to include the headers. Since the current code generator uses only one file in which all the modules are written, the headers are only written once. In the code generation, buffers are again used from a different module included as a header, since they need to implement the special synchronized read and write and the non-consuming read. There is no need to include the generation of the buffers' module in the code generator. A header will be used for that.

There are 3 types of processes which have to be used inside the execution of a program represented in DPN. Those 3 processes are **load, store** and **compute**. The module for the load processes only has to be written once, since load only happens at the beginning of the program where data is taken as an input and loaded in the buffers. For example the same load process can be executed more times in some cases, such as in the addition of 2 vectors. Each vector has to be loaded in a different buffer, but they are similar, of the same size and dimension, so there is no need for the load process to be declared twice. To avoid a load process to be declared for as many times as it is used, only the first of its occurrences leads to generating the necessary code. In the case of the other processes, all of them must be implemented in separate modules. For keeping the same order as in the DPN, each module name has an attached index, corresponding to the number it has in the DPN. This information is taken from the DPN's interface.

All the modules describing processes have the same structure: the template used for the module, followed by the name of the class or the name of the module, the public area where the ports are declared, the constructor, the implementation of the functions and the private area where other variables are declared. This structure allows the implementation of functions in the code generator which work for all the processes.

Some other functions which can be used for the templates of the modules have to be implemented. Those are generated in a separate function which is only called once. In the end, a toplevel module is generated for binding the ports of each process used and for declaring and instantiating the processes.

4.2.1 Common functions

Ports can be either input ports or output ports. The input ports come from a different DPN structure than the output ports, so separating them is straight forward by using the right structure. There are some cases in which processes use the same port multiple times. This means that the re-used port will be found for as many times in the DPN structure as it is used in the process. For the declaration part of the process, this is an issue because a port must only be declared once. This is why two set data structure are used to keep the input and output ports. After identifying the ports which have to be declared, their type has to be decided. The ports can be either buffers or FIFOs. The difference can be seen by using a DPN structure which returns true for FIFOs and false for buffers, as shown in the following code:

```
  //this example is used in the case of declaring the input ports
  switch(dpn->is_fifo[buffer_number]) {
    case true:
       cout << "  sc_core::sc_fifo_in<T> ";
       cout << "in" << buffer_number << ";" << endl;
       break;
      case false:
       cout << "  sc_core::sc_port<buffer_if<T>> ";
       cout << "in" << buffer_number << ";" << endl;
       break;
 }
```

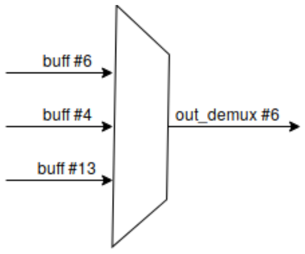
The function used for generating the templates and the name of the class is the same for all the processes. Depending on the type of the process, it generates either a type template or a type and function template. The complex templates, having both a generic type and a generic function, are applied depending on the dimension of the buffers used for output. In case of multi-dimensional buffers or arrays, an increment function is needed. This leads to the conclusion that the process using multi-dimensional buffers or arrays has templates containing both generic types and generic functions. The implementation of this behaviour is shown in the sequence of code below:

```
   //the analysis is done on the set of output ports
   set<int> out_buffers = get_out_used_buffers(dpn, p);
   complex_template = true;
   //if there exists at least one output port, the template should have both the
       generic type and the generic function
   for(const int &buffer_number : out_buffers) {
     switch(dpn->buffer_dimension[buffer_number].size()) {
        case 0:
               if(dpn->is_fifo[buffer_number]) {
              complex_template = false;
           }
           break;
        case 1:
           break;
        case 2:
           break;
     }
  }
  if(!complex_template) {
     cout << "template <typename T>" << endl;
  }
  else {
     cout << "template <typename T,\n <<
           << int (next)(int)>\n" << endl;
  }
```

The generic function for the constructor of the modules includes the name and the number of the process, as well as the name of the functions used in each module. The behaviour's implementation for each process is generated in another function. Depending on each process type, some particularities appear, so individual implementations for each process are done.

4.2.2 Particular functions

.Most of the functions are generic and do not depend on the process type as much as the processes execution's implementation depends. For example, the store module needs a particular function for the read operation because read and write instructions have to be synchronized. It is not permitted to write a value in the buffer before the value has been read from another buffer. Therefore, the read operation of the store is implemented in a different function. The read operation is represented in the DPN as a multiplexer. For example, the multiplexer used in Jacobi-1D for storing the vector used in the computation, is shown in Figure 3. All the conditions for the multiplexer's choice, the number of the ports, the counters used in the iteration through the ports are all taken from the DPN's interface. For example, the number of the buffer is given by *in_mux_buffer[p][j][k]*. At the end of the multiplexer's implementation, the synchronization measures are taken by means of event-driven mechanism provided by SystemC. An event is notified when the read is done, so the write operation can begin. The write operation is implemented in another function. This function begins with the synchronization aspects because the write operation has to wait for the read to be done. The procedure is similar to the one explained in section 4.1. Here, as well as in the read function, resources such as the counters and the number of the ports are taken form the DPN's interface.



```
//pseudocode for the multiplexer presented above
if ( -15+t___0 = 0 ) then
     out_demux #6 = buff #6
else if ( t___0 = 0 ) then
     out_demux #6 = buff #4
else if ( -1+t___0 >= 0 && 14+(-1*t___0) >= 0) then
     out_demux #6 = buff #13
```

Figure 3: Multiplexer used in Jacobi-1D

The load and compute processes only need one function each, in which the implementation is done. The decision about which buffer is used for loading the values is done in form of a demultiplexer. The conditions for the demultiplexer choice are foung in the DPN's interface. The counters which are used for iterating through the loops are taken from a special structure in the DPN's interface. The interface does not only provide the counters, but also some functions which ease the printing of the final form in SystemC code. For example, some functions of DPN's interface are able to print both the array and the counters in the C syntax.

In the function containing the store's implementation, there are performed 3 operations: reading from the buffer, executing the operation for the implemented program and writing in some other buffers. Reading is written in the form of a multiplexer, while writing is expressed as a demultiplexer. All the needed information about the inital program is interpreted from the DPN's interface. For example, a function from the DPN's interface is very helpful when the counters need to be incremented. Another DPN structure is used for printing the operation which need to be executed for each case of the compute process. The buffer and ports number, the counters, the conditions for the read and write are all taken from the DPN.

4.2.3 Toplevel module generation

The toplevel module is generated into a separate function. It contains the name of the class, the declaration of the processes and the buffers used, the instantiating of the buffers and the binding between ports and channels. For declaring the processes used in the program, it is needed to see what kind of template each process uses. If a process only has one output port, then it can implement only the type template. Otherwise, the process having many output ports has a generic type and a generic function as the template of the function. While declaring the used channels, buffers and FIFOs, some channels appear for several times, but have to be declared only once. In order to avoid this, the used channels are stored in a set. The channels' instantiating consists in giving buffers and FIFOs a size. The size is taken from DPN's interface, as shown bellow:

```
switch(dpn->buffer_dimension[buffer_number].size()) {
   case 0:
      buffer_dimension = 1;
        break;
    case 1:
        // +1 to avoid deadlocks (alternative to scanning inputs)
        buffer_dimension = dpn->buffer_dimension[buffer_number][0] + 1;
        break;
    case 2:
        k = 0;
        while (dpn->buffer_dimension[buffer_number][k]) {
            buffer_dimension = buffer_dimension *
                 dpn->buffer_dimension[buffer_number][k];
            k++;
        }
        break;
    default:
        cout << "Can't generate code for multi-dimensional buffers (yet)" << endl;
}
```

Again, this only happens once for each channel so the above mentioned set is used again. Next, the channels are bound to the ports and the main function is generated. In order to bind ports to the channels, an iteration has to be done through all the processes. For each process, the ports which are used in that process are bound to the corresponding channels. Here is an example of how the generated code of this behaviour in Jacobi-1D should look like for the process in which data is passed from one buffer to another:

```
    c5.in12.bind(b12);
    c5.out14.bind(b14);
```

In this case, buffer #12 becomes the input port and buffer #14 becomes the output port of the compute process.

5 Conclusions and further development

5.1 Technical Conclusion

As mentioned previously, there has been noticed the need of optimization in hardware and software programming due to the increasing level of parallelism in supercomputers and in classical end-user computers. The CASH team works on improving the way in which high-performance computing is handled. The CASH team wants to create a flow in which a classical written C-code is synthesized and mapped on the FPGA. Parallelism is extracted into DPN (data-aware process network) by using the dcc-light compiler. Synthesis of the parallel code, as well as simulation are two important steps of the research. Simulation of the parallel code is especially useful for debugging of dcc-light. For this, a code generator which can output SystemC code used in simulation, based on the parallel representation extracted into DPN.

There are a few aspects which should be improved in the code generator implemented during this internship. One thing consists in trying to extract the datatype from the DPN representation. Another thing is related to synchronization, which can be improved in the SystemC code. Optimizations on the current implementation of operations on buffers and the functions of the store process can be done from the synchronization point of view.

5.2 Personal Conclusion

I think that for me, an undergraduate student, the summer internship at ENS-Lyon in the CASH team was a great opportunity. I am currently trying to decide my career path, so I think the experience of working here was a very productive one. It helped me get an inside perspective of how work is performed in the research field.

Even though I only was part of the CASH team for a short period of about 2 months, I learnt a lot of things from them. From knowledge in hardware programming to how to write a scientific report, I gained experience in professional matters and improved some skills I had before. I developed some skills such as working devotedly in a team, but also doing tasks on my own, being punctual and serious about my work and being able to understand someone else's previously done work. It was nice to witness their effort in developing useful solutions by their research work. I can tell now for sure that being committed to your work and trying to improve it daily with small steps leads to successful results in the end.

It has been an honor to be part of this team of professionals and I am thankful to the CASH team and the institute for giving me the chance to do my internship among them. I am sure that all the knowledge gained during this internship will be useful in my future career.