Implementation of a centralized scheduler for the Mitrion Virtual Processor

Magnus Persson

LiTH-ISY-EX--08/4178--SE

Linköping 2008
Implementation of a centralized scheduler for the Mitrion Virtual Processor

Examensarbete utfört i Datorteknik vid Tekniska högskolan i Linköping av

Magnus Persson

LiTH-ISY-EX--08/4178--SE

Handledare: Olle Seger
           ISY, Linköpings universitet
           Pex Tufvesson
           Mitrionics

Examinator: Olle Seger
            ISY, Linköpings universitet

Linköping, 9 December, 2008
Titel
Implementation av en centraliserad skedulerare för Mitrion Virtual Processor

Författare
Magnus Persson

Sammanfattning
Mitrionics is a company based in Lund, Sweden. They develop a platform for FPGA-based acceleration, the platform includes a virtual processor, the Mitrion Virtual Processor, that can be custom built to fit the application that is to be accelerated. The purpose of this thesis is to investigate the possible benefits of using a centralized scheduler for the Mitrion Virtual Processor instead of the current solution which is a distributed scheduler. A centralized scheduler has been implemented and evaluated using a set of benchmark applications. It has been found that the centralized scheduler can decrease the number of registers used to implement the Mitrion Virtual Processor on an FPGA. The size of the decrease depends on the application, and certain applications are more suitable than others. It has also been found that the introduction of a centralized scheduler makes it more difficult for the place and route tool to fit a design on the FPGA resulting in failed timing constraints for the largest benchmark application.
Abstract

Mitrionics is a company based in Lund, Sweden. They develop a platform for FPGA-based acceleration, the platform includes a virtual processor, the Mitrion Virtual Processor, that can be custom built to fit the application that is to be accelerated. The purpose of this thesis is to investigate the possible benefits of using a centralized scheduler for the Mitrion Virtual Processor instead of the current solution which is a distributed scheduler. A centralized scheduler has been implemented and evaluated using a set of benchmark applications. It has been found that the centralized scheduler can decrease the number of registers used to implement the Mitrion Virtual Processor on an FPGA. The size of the decrease depends on the application, and certain applications are more suitable than others. It has also been found that the introduction of a centralized scheduler makes it more difficult for the place and route tool to fit a design on the FPGA resulting in failed timing constraints for the largest benchmark application.

Sammanfattning

Acknowledgments

I would like to thank my supervisor Pex Tufvesson and the staff at Mitrionics for all their help and support. I would also like to thank my supervisor and examiner Olle Seger at the Division of Computer Engineering at the Department of Electrical Engineering at Linköping University for helpful input and ideas.
Contents

1 Introduction 3
  1.1 Background ................................................. 3
    1.1.1 Mitrionics ........................................... 4
    1.1.2 The Mitrion Virtual Processor ...................... 4
    1.1.3 The Mitrion Platform ............................... 4
  1.2 Problem Specification ...................................... 5
  1.3 Purpose .................................................. 5
  1.4 Method .................................................. 5
  1.5 Limitations ............................................... 5
  1.6 Thesis Outline ............................................ 6
  1.7 Glossary ................................................ 6

2 FPGA 9
  2.1 CLB ...................................................... 9
  2.2 LUT ....................................................... 9
  2.3 Switch Matrix ............................................. 10
  2.4 Place and Route ......................................... 11

3 The Centralized Scheduler 13
  3.1 Processing Elements ....................................... 14
    3.1.1 NM ................................................... 14

4 Implementation 15
  4.1 Transformations ........................................... 15
    4.1.1 M2Z .................................................. 16
    4.1.2 Z2M .................................................. 16
    4.1.3 NMVC ............................................... 16
    4.1.4 Copy ............................................... 18
    4.1.5 VectMake .......................................... 18
    4.1.6 VectSplit .......................................... 19
    4.1.7 Back ............................................... 19
  4.2 Optimizations ............................................ 20
    4.2.1 Z2M-M2Z ............................................ 20
    4.2.2 NMVC clouds ....................................... 20
    4.2.3 NMVC with 1 input and 1 output .................. 23
List of Figures

1.1 Mitrion ....................................................... 5
2.1 Interconnect network ........................................ 10
2.2 Possible connections to a connect matrix. .................. 10
2.3 Switch matrix. .............................................. 11
3.1 The NM node ADD. ........................................... 14
4.1 A graph with two processing elements of type NM before transfor-
mations. ..................................................... 15
4.2 The M2Z node. .............................................. 16
4.3 The Z2M node. .............................................. 16
4.4 The NMVC node. ............................................ 17
4.5 The graph from Figure 4.1 after the transformations. ........ 17
4.6 Copy transform. ............................................ 18
4.7 VectMake transform. ........................................ 18
4.8 VectSplit transform. ....................................... 19
4.9 Back-M2Z transform. ...................................... 19
4.10 Back-Z2M transform. ...................................... 20
4.11 Z2M-M2Z optimization. .................................... 20
4.12 Z2M-M2Z optimization on the graph from Figure 4.5 ....... 21
4.13 NMVC optimization. ....................................... 21
4.14 The graph from Figure 4.1 after transformations and optimizations. 22
4.15 NMVC node with one input and one output. ................. 23
5.1 FPGA editor with blast loaded on a Virtex4 vlx200. Blue CLBs are occupied and grey are unused. ............. 26
7.1 Data dependency graph for the Permutation application. .... 30
7.2 Data dependency graph for the Permutation application after im-
plementation of a centralized scheduler ....................... 31
7.3 Data dependency graph for the Shift application .............. 32
7.4 Data dependency graph for the Shift application after implement-
ation of a centralized scheduler ........................... 34
7.5 Data dependency graph for the Ludesi application, without central scheduling on the left and with central scheduling on the right. 36
7.6 Data dependency graph for the Blast application. With distributed scheduling on the left and centralized scheduling on the right. .... 38

List of Tables

2.1 LUT for a two input AND gate ............................. 9
7.1 Register report ............................................. 30
7.2 Timing summary ............................................. 31
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.3 Logic utilization and distribution</td>
<td>31</td>
</tr>
<tr>
<td>7.4 Register report</td>
<td>33</td>
</tr>
<tr>
<td>7.5 Timing summary</td>
<td>34</td>
</tr>
<tr>
<td>7.6 Logic utilization and distribution</td>
<td>34</td>
</tr>
<tr>
<td>7.7 Register report</td>
<td>35</td>
</tr>
<tr>
<td>7.8 Timing summary</td>
<td>35</td>
</tr>
<tr>
<td>7.9 Logic utilization and distribution</td>
<td>35</td>
</tr>
<tr>
<td>7.10 Register report</td>
<td>37</td>
</tr>
<tr>
<td>7.11 Timing summary</td>
<td>37</td>
</tr>
<tr>
<td>7.12 Logic utilization and distribution</td>
<td>39</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The need for computational power is ever increasing as well as the need for power conservation in the form of reducing the amount of electricity we consume. Conventional supercomputers and servers used by companies and other institutions provide large computational power but they also consume large amounts of energy. The solution for certain companies and scientists is spelled FPGA-based acceleration. For more information about FPGAs see the glossary or Chapter 2. This means that you accelerate a specific part of your application, or the whole application depending on size, in an FPGA. The application is implemented in hardware, in this case an FPGA, which runs faster than if it was done in software and executed on a conventional processor. The way it has been done is by assigning the task of designing the hardware to hardware engineers with knowledge in hardware description languages, like VHDL or Verilog. This is expensive and time consuming. A easier way of doing this is letting your software developers write the application in a higher-level language, and then handing it over to an automatic tool that compiles it into hardware, in this case a bitfile used to program the FPGA circuit. This is what Mitrionics have done with their Mitrion platform. This makes hardware acceleration accessible to scientists and companies that do not possess the hardware engineering skills to design an accelerator themselves but would still benefit from accelerating their applications. An FPGA-based solution running the Mitrion Virtual Processor typically does the job 10-100 times faster that a conventional CPU while at the same time consuming one quarter of the energy a standard CPU does.

This chapter contains an introduction to the thesis. It will give a background to the thesis, along with a problem specification and a presentation of the purpose.

1.1 Background

This section contains a presentation of the company at which this thesis is carried out and an introduction to their product The Mitrion Platform.
1.1.1 Mitrionics

Mitrionics is a company based in Lund, Sweden. It was founded in 2001 and currently has 24 employees [1]. The company develops the Mitrion Software Acceleration Platform which is used for FPGA-based processing. The platform enables easier and faster programming of FPGAs without requiring any knowledge in hardware design or hardware description languages as VHDL or Verilog. The company also sells a series of computer systems called Mitrionics MVP Hybrid Computing Systems, which are computer systems equipped with FPGA modules for use with the Mitrion Software Acceleration Platform.

1.1.2 The Mitrion Virtual Processor

The Mitrion Virtual Processor, MVP, is a massively parallel processor with a special architecture. The processor architecture can be compared to a set of tiles, all tiles fit together and depending on how they are placed, the set of tiles can perform different functions. This enables the Mitrion Virtual Processor to be custom built to suit the algorithm in question. This typically results in a 10-100 times speedup compared to when the algorithm is executed on a conventional processor. The speedup is limited to how well suited the algorithm is for parallel execution, as well as the size of the FPGA. This means that an increase in FPGA size often can be translated into an increase in computational power.

1.1.3 The Mitrion Platform

The Mitrion Platform consists of the Mitrion Virtual Processor and the Mitrion Software Development Kit. Included in the SDK is a C-type language called Mitrion-C developed by Mitrionics which is used for writing the programs for the Mitrion Virtual Processor. Mitrion-C is developed for parallel programming and helps the programmer to harness the possibilities of parallel execution given by the Mitrion Virtual Processor.

Another advantage of using an FPGA-based solution running the Mitrion Virtual Processor, except the increased performance, is reduced power consumption. A large FPGA running the MVP consumes at most 25W, a fast conventional processor consumes up to 100W. At 15 times speedup, the FPGA with the MVP solution would only consume 2 percent of the power used by the conventional processor to get the job done.[1]

The background to accelerating a software function in hardware is the 90/10 law, it says that 10 percent of the code is executed 90 percent of the time.[5] So if the critical section of the code, being run 90 percent of the time, is identified and executed on dedicated hardware, in this case the Mitrion Virtual Processor, the performance of the whole program can be increased.
1.2 Problem Specification

In today’s implementation of the Mitrion Virtual Processor, the scheduling mechanism is a distributed state machine, where every processing element handles its own scheduling. This results in a scalable, distributed solution that handles classical hardware engineering tasks such as fanout and timing quite well. However, there are algorithm constructs that will gain from using a centralized scheduler with top down global knowledge about processing element structures. Groups of processing elements schedulers are then allowed to be optimized (much like any synthesis program can do with combinatorial logic).

1.3 Purpose

The purpose of this Master’s thesis is to investigate the possible gains of using a centralized scheduler instead of a distributed scheduler for the Mitrion Virtual Processor, and implement such a centralized scheduler for all types of algorithms.

1.4 Method

In the beginning of the thesis a suitable type of processing element will be identified. Then a centralized scheduler will be implemented for those type of processing elements. The benefit of using the centralized scheduler will be evaluated using a set of benchmark applications, it will be measured in the number of flip flops used to implement the Mitrion Virtual Processor and compared to the current scheduler.

1.5 Limitations

This thesis will not investigate the possible gains of using a centralized scheduler for all type of processing elements. It will focus on one group of processing
elements and look into what possible benefits there might be from using a central-
ized scheduler for them while leaving the remaining processing elements to the
distributed scheduling mechanism.

1.6 Thesis Outline

Chapter 1 contains a short introduction to the thesis.

Chapter 2 contains a short introduction to the building blocks of an FPGA.

Chapter 3 explains the idea and method of introducing a central scheduler to
the MVP.

Chapter 4 covers the implementation of the central scheduler.

Chapter 5 covers the aspect of timing for the MVP.

Chapter 6 contains the verification of the central scheduler.

Chapter 7 introduces the benchmark applications and presents the results from
them.

Chapter 8 contains the results of the thesis.

1.7 Glossary

Explanations of words and abbreviations used in this thesis.

CLB Configurable Logic Block

HDL Hardware Description Language.

HPC High-performance computing.

LUT Lookup table.

M domain Mitrion domain, the domain between a Z2M node and a M2Z node.
This part is not subjected to central scheduling.

M2Z A node that converts from M to Z domain.

MVP Mitrion Virtual Processor.

NM The type of nodes that will be subjected to centralized scheduling.

NMVC Processing Element Scheduler (the centralized scheduler).

Node Equivalent to PE, used interchangeable in this document.

FPGA Field-Programmable Gate Array.

Verilog A hardware description language.
VHDL  A hardware description language.

PAR  Place And Route, the process of mapping logic to a FPGA.

PE  Processing Element, the building blocks which make up the Mitrion Virtual Processor.

Processing element The building blocks which make up the Mitrion Virtual Processor.

SDK  Software Development Kit.

Synthesis  Convert a high-level design, VHDL, to a low-level design, logic gates.

Z domain  The domain between a M2Z node and a Z2M node. This is subjected to central scheduling.

Z2M  A node that converts from Z to M domain.
Chapter 2

FPGA

The acronym FPGA stands for Field Programmable Gate Array. An FPGA is a programmable logic device. The function of the device can be described using a hardware description language like VHDL or Verilog. The main vendors of FPGAs are Xilinx and Altera. The target FPGA for the synthesis done during the work on this thesis is a Xilinx Virtex-4 and the information contained in this chapter applies to FPGAs from Xilinx. Different vendors have different FPGA architectures.

2.1 CLB

The configurable logic block, CLB, is the basic building block of the FPGA. It consists of slices, 4 slices per CLB in a Xilinx Virtex-4. Each slice consists of two LUTs, two storage units, multiplexers, carry logic and arithmetic gates.

2.2 LUT

A look-up table, LUT, is a function generator capable of implementing any boolean function with up to four inputs. The multiplexers present in the slices can be combined to form boolean function with more than 4 inputs.

<table>
<thead>
<tr>
<th>x0</th>
<th>x1</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
The propagation delay through the LUT is the same regardless of the complexity of the boolean function.[6]

2.3 Switch Matrix

The CLBs must be connected to each other and that is done via a programmable interconnect network[4]. Each CLB is surrounded with a connect matrix on each side and a switch matrix at every corner. The inputs and outputs to the CLB are connected to a connect matrix and the connect matrixes are connected together via the switch matrixes. See Figure 2.1 for an overview of the programmable interconnect network, the boxes marked with a C are connect matrixes and the boxes marked with a S are switch matrixes. Figure 2.2 shows the possible connections in a connect matrix and Figure 2.3 shows the possible connections in a switch matrix.

Figure 2.1. Interconnect network

Figure 2.2. Possible connections to a connect matrix.
2.4 Place and Route

Place and route is a step in designing a circuit, in this case it is the process of mapping logic onto an FPGA chip. It is comprised of two steps, the first one is deciding where to place the logic elements on the chip, and the second step is to route all wires so the logic elements are connected. All this has to be done while keeping in mind that there are a limited amount of space on a FPGA, in the form of CLBs, interconnect grid and memories. There are also often a number of timing constraints that needs to be meet if the circuit is to function correctly. Because of the complexity of circuits that can fit into an FPGA the process of PAR is often done by an automatic tool, often supplied by the vendor of the FPGA.

Figure 2.3. Switch matrix.
Chapter 3

The Centralized Scheduler

The Mitrion Virtual Processor has a special architecture, it can be described as a set of tiles, or processing elements which is another name for the same thing. Each type of tile perform a different function, for example addition, and all tiles fit together. Depending on how the tiles are put together they perform a different function, they form a different configuration of the MVP. The MVP configuration can be viewed as a data dependency graph were the tiles are represented by nodes in the graph. Each tile has its own scheduling mechanism, this is the distributed scheduling mechanism mentioned earlier in the problem specification. This distributed scheduling mechanism is what allows the nodes to be put together in different arrangements and still work together and perform a function.

Scheduling in this context is deciding whether or not a node should execute. Currently this is decided by the distributed scheduler at every node. The distributed scheduler looks at the control signals from nodes connected at the inputs and the outputs of the node. For an ADD node, the requirement is that valid data should be present at the inputs and the data previously generated at the output should have been taken care of by the node below. When the ADD node has executed it notifies the surrounding nodes via the control signals that it has used the data at the inputs and produced data at the output. Another node might be waiting on the data from the ADD node, which now enables that node to execute, or perhaps a node above is waiting for the ADD node to use the data it has produced so it can execute again.

The idea of the centralized scheduler is to break out the control logic from each node into a node which will be called NMVC, which will act as our centralized scheduler. Each node that we want to schedule with a centralized scheduler gets a centralized scheduler that only is responsible for that node. A special node called M2Z is placed above our node, this nodes marks a beginning to a partition of the data dependency graph that will be handled by a centralized scheduler. Below our node a node called Z2M is placed to mark the end of the section handled by the centralized scheduler. The node we wanted to add central scheduling to now has one node above, one on the side and one below, these nodes encapsulates the node, and the scheduling of the node is now handled by the NMVC node. The process
of adding the nodes to the data dependency graph is explained in Chapter 4.1. Since there now is one centralized scheduler present at every node, what we have now is basically the same solution as before, a distributed scheduler.

The next step is to identify which centralized schedulers can be merged together and then schedule more than one node and thus forming one centralized scheduler. How this is done is described in Chapter 4.2. After all NMVC nodes that can be merged has been merged, there will be several centralized schedulers scheduling different parts of the graph. The only time there will be only one centralized scheduler in the graph is when the Mitrion Virtual Processor has been configured for a very small application.

### 3.1 Processing Elements

One type of processing elements, also called nodes, have been chosen for centralized scheduling. That type is NM nodes. They have been chosen because they are the largest class of nodes and their behavior in regards to scheduling is the same. This makes them an ideal candidate for centralized scheduling and thus they are the type of nodes that will be subjected to centralized scheduling.

#### 3.1.1 NM

A node is classified as a NM node if it uses the data on all inputs and generates data on all outputs. This is the largest class of nodes and they have been chosen for centralized scheduling based on their predictable behavior. A NM node may have an arbitrary number of inputs and outputs. An example of a NM node is the processing element ADD, see Figure 3.1.

![ADD node](image)

*Figure 3.1. The NM node ADD.*

An ADD node perform the boolean function \( a = b + c \). 'a' can not be calculated until both 'b' and 'c' has become available. When processing a list of numbers, i.e. \( an = bn + cn \) care must be taken that the correct element in the b sequence is added to the correct element in the c sequence.
Chapter 4

Implementation

4.1 Transformations

The first step in the implementation of a centralized scheduler for the MVP is to perform a number of transformations on the data dependency graph, which is our representation of the MVP. The aim of the first set of transformations is to introduce the node called NMVC. This is our centralized scheduler. This is done by rerouting the control signals so they pass through the NMVC node and not through the distributed scheduler present at every node. The first step is to identify the nodes of the type NM. This is done by simply traversing through the graph and for each node of type NM we find we introduce the nodes M2Z, Z2M and NMVC. Figure 4.1 shows a graph with two ADD nodes before transformation.

A node of type M2Z converts the existing data format to a format that gives us access to the control signals. We can then route them to the NMVC. The Z2M is the node that converts the signals that were split up by the M2Z back to the original format. This enables the distributed scheduler to take over until another M2Z marks the beginning of another block that is scheduled by a centralized scheduler, NMVC. The three nodes, NMVC, M2Z and Z2M, introduced in this chapter are all of type NM which is the same type as the nodes that we will now subject to centralized scheduling.

![Figure 4.1. A graph with two processing elements of type NM before transformations.](image-url)
4.1.1 M2Z

This node has one input and two outputs. A node of this type is placed on every input of every NM node that should be scheduled by the centralized scheduler. This node splits the original data format into a datapath and a control path. One of the outputs is connected to the NM node and the other output is connected to the NMVC node. The input is connected to the node above. Figure 4.2 shows a M2Z node.

![M2Z Diagram](image)

Figure 4.2. The M2Z node.

4.1.2 Z2M

This node has two inputs and one output. A node of this type is placed on every output of every NM node that should be scheduled by the centralized scheduler. This node merges the datapath and control path into the original data format. One of the inputs is connected to the NM node and the other input is connected to the NMVC node. The output is connected to a node below. Figure 4.3 shows a Z2M node.

![Z2M Diagram](image)

Figure 4.3. The Z2M node.

4.1.3 NMVC

A NMVC node is a processing element scheduler, the centralized scheduler, it is placed alongside every NM node in the graph that should be scheduled by the centralized scheduler. It contains the logic for the scheduling. It can have an arbitrary number of inputs and outputs. The inputs are connected to M2Z nodes above and the outputs are connected to Z2M nodes below. Figure 4.4 displays a NMVC node. Figure 4.5 displays the graph from Figure 4.1 after transformation, plain black connectors represent the original Mitron format, dashed black connectors represents data only and solid green lines represent control signals.
4.1 Transformations

Figure 4.4. The NMVC node.

Figure 4.5. The graph from Figure 4.1 after the transformations.
4.1.4 Copy

A copy node copies the data on the input to n number of outputs, this is an existing node, not implemented during the work on this thesis. This transform transforms the copy node into a tree structure. See Figure 4.6. Transformations on copy nodes is not a part of the centralized scheduler but it gives the compiler more options where to insert buffers into combinatorial paths, thus making it easier for the place and route tool to do its job and conform to timing constraints.

![Figure 4.6. Copy transform.](image)

4.1.5 VectMake

A VectMake node creates a vector from the inputs, this is an existing node, not implemented during the work on this thesis. This transform transforms the VectMake node into a tree structure. Two VectMake nodes and one VectJoin node are used to merge the two vectors. See Figure 4.7. Transformations on VectMake nodes is not a part of the centralized scheduler but it gives the compiler more options where to insert buffers into combinatorial paths, thus making it easier for the place and route tool to do its job and conform to timing constraints.

![Figure 4.7. VectMake transform.](image)
4.1.6 VectSplit

A VectSplit node splits a vector into n number of vectors, this is an existing node, not implemented during the work on this thesis. This transform transforms the VectSplit node into a tree structure. See Figure 4.8. Transformations on VectSplit nodes is not a part of the centralized scheduler but it gives the compiler more options where to insert buffers into combinatorial paths, thus making it easier for the place and route tool to do its job and conform to timing constraints.

![Figure 4.8. VectSplit transform.](image)

4.1.7 Back

The back node is used to cut combinatorial paths, this type of node is inserted after the centralized scheduler has been introduced. This can result in back nodes being placed on arcs in the Z domain, the domain handled by the centralized scheduler, and thus they have to be moved into the M domain, the domain handled by the distributed scheduler. This transformation does just that, see Figure 4.9 and 4.10 for illustrations of the two possible cases. This transformation is not a part of the centralized scheduler, it counteracts a side effect of the introduction of the centralized scheduler.

![Figure 4.9. Back-M2Z transform.](image)
4.2 Optimizations

This is the step where the centralized schedulers that where introduced in the previous chapter are merged together to schedule more than one node each. This is done with a few optimizations to the data dependency graph. It is done in two steps, the first step is to find Z2M nodes connected to M2Z nodes. A construct like that indicates that two partitions of the graph that are scheduled by a centralized scheduler are connected to each other. Therefore the Z2M and M2Z nodes can be removed and we have merged two areas which were scheduled by centralized schedulers into one area. See Figure 4.12. The second step is to merge the two central schedulers into one. See Section 4.2.2.

4.2.1 Z2M-M2Z

This optimization merges different areas that are subjected to centralized scheduling and thus allowing NMVC nodes to be merged together, Section 4.2.2 describes merging of NMVC nodes. The graph is traversed and if a Z2M followed by a M2Z is discovered, they are simply removed. Figure 4.11 illustrates the optimization and Figure 4.12 shows how it is applied to the graph from Figure 4.5 and how the area above the Z2M node is merged with the area below the M2Z node.

4.2.2 NMVC clouds

After the optimization of the Z2M and M2Z nodes, graph constructs as those in Figure 4.12 and Figure 4.13 can be found in the graph. It is NMVC nodes connected to each other. They can be merged together to a single NMVC. The graph
is traversed and for each NMVC found, the outputs are checked for connections to NMVC’s. If a NMVC is found on an output, the two NMVC’s are merged together. Figure 4.14 shows the results on the graph from Figure 4.12 after merging two NMVC nodes. The two adders in the graph can now be scheduled by the centralized scheduler as if they were one node.
Figure 4.14. The graph from Figure 4.1 after transformations and optimizations.
4.3 Java Simulation Models

4.2.3 NMVC with 1 input and 1 output

Another optimization of NMVC nodes that can be done is removal of NMVC nodes which have only one input and one output. This is shown in Figure 4.15.

Figure 4.15. NMVC node with one input and one output.

4.3 Java Simulation Models

An MVP can be simulated in a GUI graph, this is a high level simulation of the MVP and it will be discussed in Chapter 6.1. In order for the simulation to work with the nodes that have been introduced in Section 4.1.1, 4.1.2 and 4.1.3, simulation models for each node has to be implemented. The simulation model for a node describes the nodes behavior during simulation. For a node of type NM to execute, data on all inputs must be present. See Appendix A for VHDL models.

4.3.1 M2Z

The simulation model for the node M2Z copies the data from the input to the left output port while outputting a boolean with the value true on the right output port. See Section 4.1.1 for more information about the M2Z node.

4.3.2 Z2M

The simulation model for the node Z2M copies the data from the left input to the left output while discarding the boolean with value true that is present at the right input. See Section 4.1.2 for more information about the Z2M node.

4.3.3 NMVC

The simulation model for the node NMVC generates booleans with value true on all outputs when it executes, assuming there are are booleans with value true on all inputs. Otherwise it does not execute. See Section 4.1.3 for more information about the NMVC node.
Chapter 5

Timing

The current distributed scheduling mechanism handles timing quite well. During compilation buffers are placed in between processing elements to break up combinatorial paths that might be too long. This ensures that the processor configuration generated meets the timing constraints. This differs when the centralized scheduler is introduced since the buffers are inserted into the graph before the centralized schedulers are inserted. There will be no buffers in the Z-domain, which is a partition of the graph scheduled by a centralized scheduler.

5.1 Timing Model

In order for the compiler to assemble a processor configuration that will meet timing requirements, timing models for the three new nodes must be created. The compiler will use the timing model for each node to create a timing graph, based on that graph the compiler will determine where to insert registers to break up combinatory paths that have been found to be too long. Since the scheduling mechanism in the scheduler node NMVC is the same as in the distributed scheduler, the timing model for the distributed scheduler can be reused in the central scheduler. However the timing model for the datapath is different, there are no datapaths between the inputs and no datapaths between the inputs and outputs, which means that the latency on the datapath through the NMVC node will be reported as -1, which indicates that there is no path.

The other two nodes, M2Z and Z2M are scheduled with their built in distributed scheduler and thus there is no need to implement a timing model for the scheduling mechanism in them. A timing model for the datapath through them must be implemented though. Both of the nodes copies the data on input0 to output0, that is an operation with low combinatorial delay and consequently the combinatorial delay is set to zero.
5.2 Timing Constraints

The Mitrion Virtual Processor is clocked at 100MHz which gives us a maximum delay of 10ns. The timing constraints for the device are set accordingly.

5.3 BLAST

The synthesis of the BLAST application described in Section 7.4 reports that the PAR tool was not able to meet all timing constraints. The synthesis tool reports a number of paths as being too long, there are mainly paths between on-chip memories and logic. Figure 5.1 shows a screenshot of an application called FPGA editor which has loaded the design file for the BLAST application. The Figure shows the layout of the logic on the FPGA in blue. One path that is too long has been marked with the color red in the figure.

Figure 5.1. FPGA editor with blast loaded on a Virtex4 vlx200. Blue CLBs are occupied and grey are unused.
Chapter 6

Verification

Verifying the functional correctness after the implementation of the centralized scheduler is done in three steps. The first step is to use the Mitrion simulator. See Section 6.1 for information about the simulator. The benchmark algorithm is compiled along with an input file and run in the simulator. The output is then compared to the output of a simulation run with the original scheduler. If the outputs are the same, the function of the program is correct. This step tests the simulation models implemented for the central scheduler. The next step in verifying the functional correctness of the program is to use the HDL simulator Riviera. See Section 6.2. Riviera is used to verify that the VHDL code for the centralized scheduler generated by the compiler is correct. The third step is actually running the resulting MVP in an FPGA for final verification of the function. It will be run on a SGI RASC RC100. See section 6.3 for more information about RC100.

6.1 The Mitrion Simulator

The Mitrion SDK includes a simulator. The simulator can be used to verify the functionality of your Mitrion-C program and analyze the performance. The simulator can be run in either a batch mode or a GUI mode, the differences between them are covered in Section 6.1.1 and Section 6.1.2. Both modes will be used in this master thesis to verify the functional correctness after the implementation of the centralized scheduler.

6.1.1 Graphical User Interface Mode

When a Mitrion-C program is compiled with the GUI option a data dependency graph will be generated. An example of a data dependency graph can be seen in Figure 7.1. The graph displays data dependencies, it also displays the parallelism, both data and pipeline parallelism. The nodes in the graph represent operations and the edges represent data dependencies. Nodes are dependent on nodes above unless the edge is yellow which represent reversed dependencies.
6.1.2 Batch Mode

In batch mode the simulator executes somewhat of a low level simulation of the program which closely resembles the Mitrion Virtual Processor that will be run on the FPGA. This makes the batch mode good for performance measurements.

6.2 Riviera PRO

Riviera is a mixed language HDL simulator. In this thesis Riviera is used to verify that the VHDL code generated by the compiler behaves correctly. This is done by supplying a flag to the Mitrion compiler telling it to generate a testbench, which can then be used in Riviera. The same input file as used for the Mitrion simulator are used here. A simulation in Riviera is carried out and the results are compared to the results from the Mitrion simulator. If the results are the same, the hardware function is correct.

6.3 SGI RASC RC100

SGI is a company in the high-performance computing, HPC, business. They have developed a computer system called RASC RC100 where RASC stands for Reconfigurable Application Specific Computing and RC100 being the name of the blade server. The system contains two Virtex 4 LX200 FPGAs. This is one of the target systems when synthesizing the benchmark applications.

6.4 FPGA editor

FPGA editor is a part of Xilinx ISE package, it is used to configure and visualize field programmable gate arrays. The application requires an .ncd netlist file which contains your logic and how it is mapped to logic blocks in the FPGA. The application gives you among other things an opportunity to manually place and route components before handing it over to the automatic PAR tool. In this thesis it is used to visualize timing paths. See Figure 5.1 for an example of how the FPGA editor application can display a design.
Chapter 7

Benchmark Applications

The benefits of the centralized scheduler will be evaluated using a set of benchmark applications. This chapter will give an introduction to the applications along with the results from the synthesis of each program. Each program is synthesized twice, one time with the centralized scheduler and one time with the original distributed scheduler. The results of the synthesis is then compared in Chapter 8 to evaluate the centralized scheduler. The target FPGA is a Xilinx Virtex-4 LX200.

7.1 Permutation

The application copies and flips a vector. The Mitrion-C code for the algorithm is shown in Listing 7.1. The algorithm takes an unsigned integer as input. It casts the uint to bits and then casts it to a vector of length 16, see row 5 and 6. Two new vectors are created as permutated copies of the input vector, see row 8 and 11. The two new vectors are then cast back to the input type, unsigned integer and returned. This is the algorithm that captures the essence of the centralized scheduler. It shows the type of program constructs that will benefit from using a centralized scheduler. This is a common construct in bigger applications, for example the BLAST application described in Section 7.4. The data dependency graph for this algorithm using the distributed scheduling mechanism is shown in Figure 7.1. The data dependency graph for this algorithm using the centralized scheduler is shown in Figure 7.2. One can easily see that all the yellow nodes that are there to break up combinatorial paths can be removed when using the centralized scheduler, hence the dramatic decrease in number of registers used to implement this design.
Listing 7.1. Permutation

```c
Mitrion-C 1.2;
main(uint:64 a_uint)
{
    bits:64 a_bits = a_uint;
    uint:4[16] a = a_bits;
    uint:4[16] b1 = [a[15], a[14], a[13], a[12], a[11],
                    a[10], a[9], a[8], a[7], a[6], a[5],
                    a[4], a[3], a[2], a[1], a[0]];
    uint:4[16] b2 = foreach(i in [0..15])
    {
        b = a[15-i];
    } b;
    bits:64 b1_bits = b1;
    uint:64 b1_uint = b1_bits;
    bits:64 b2_bits = b2;
    uint:64 b2_uint = b2_bits;
} (b1_uint, b2_uint);
```

Figure 7.1. Data dependency graph for the Permutation application.

Table 7.1. Register report

<table>
<thead>
<tr>
<th></th>
<th>Flip-Flops</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributed</td>
<td>484</td>
</tr>
<tr>
<td>Centralized</td>
<td>4</td>
</tr>
</tbody>
</table>
Figure 7.2. Data dependency graph for the Permutation application after implementation of a centralized scheduler

Table 7.2. Timing summary

<table>
<thead>
<tr>
<th>Scheduling</th>
<th>Min period [ns]</th>
<th>Max frequency [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributed</td>
<td>5.993</td>
<td>166.856</td>
</tr>
<tr>
<td>Centralized</td>
<td>1.975</td>
<td>506.239</td>
</tr>
</tbody>
</table>

Table 7.3. Logic utilization and distribution

<table>
<thead>
<tr>
<th>Scheduling</th>
<th>Slice flip-flops</th>
<th>4 input LUTs</th>
<th>Occupied slices</th>
<th>Gate count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributed</td>
<td>419</td>
<td>628</td>
<td>474</td>
<td>7712</td>
</tr>
<tr>
<td>Centralized</td>
<td>4</td>
<td>8</td>
<td>7</td>
<td>80</td>
</tr>
</tbody>
</table>
7.2 Shift

This application takes a 256 bit value and a 7 bit offset as input. The value is right shifted as many steps as the offset states. A shift-tree is created. The pseudo code for the algorithm is shown in Algorithm 1. The benefit of using the centralized scheduler for a algorithm like this is nearly zero when looking a the number of registers needed to implement this design, see Table 7.4. This comes from the fact that this algorithm lacks the program constructs found in the Permutation application. Figure 7.3 shows the Shift application using the distributed scheduler and Figure 7.4 shows the application when using the centralized scheduler.

![Data dependency graph for the Shift application](image_url)
Algorithm 1 Shift

1: \textbf{if} \text{offvec}[6] \textbf{then}
2: \quad \text{bits:192 } i_1 \leftarrow i_0 \gg 64
3: \textbf{else}
4: \quad \text{bits:192 } i_1 \leftarrow i_0
5: \textbf{end if}
6: \textbf{if} \text{offvec}[5] \textbf{then}
7: \quad \text{bits:160 } i_2 \leftarrow i_1 \gg 32
8: \textbf{else}
9: \quad \text{bits:160 } i_2 \leftarrow i_1
10: \textbf{end if}
11: \textbf{if} \text{offvec}[4] \textbf{then}
12: \quad \text{bits:144 } i_3 \leftarrow i_2 \gg 16
13: \textbf{else}
14: \quad \text{bits:144 } i_3 \leftarrow i_2
15: \textbf{end if}
16: \textbf{if} \text{offvec}[3] \textbf{then}
17: \quad \text{bits:136 } i_4 \leftarrow i_3 \gg 8
18: \textbf{else}
19: \quad \text{bits:136 } i_4 \leftarrow i_3
20: \textbf{end if}
21: \textbf{if} \text{offvec}[2] \textbf{then}
22: \quad \text{bits:132 } i_5 \leftarrow i_4 \gg 4
23: \textbf{else}
24: \quad \text{bits:132 } i_5 \leftarrow i_4
25: \textbf{end if}
26: \textbf{if} \text{offvec}[1] \textbf{then}
27: \quad \text{bits:130 } i_6 \leftarrow i_5 \gg 2
28: \textbf{else}
29: \quad \text{bits:130 } i_6 \leftarrow i_5
30: \textbf{end if}
31: \textbf{if} \text{offvec}[0] \textbf{then}
32: \quad \text{bits:128 } i_7 \leftarrow i_6 \gg 1
33: \textbf{else}
34: \quad \text{bits:128 } i_7 \leftarrow i_6
35: \textbf{end if}
36: \textbf{return } i_7

\begin{table}[h]
\centering
\caption{Register report}
\label{tab:register_report}
\begin{tabular}{|l|c|}
\hline
\textbf{Scheduling} & \textbf{Flip-Flops} \\
\hline
Distributed & 279 \\
Centralized & 280 \\
\hline
\end{tabular}
\end{table}
Figure 7.4. Data dependency graph for the Shift application after implementation of a centralized scheduler

Table 7.5. Timing summary

<table>
<thead>
<tr>
<th>Scheduling</th>
<th>Min period [ns]</th>
<th>Max frequency [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributed</td>
<td>4.829</td>
<td>207.084</td>
</tr>
<tr>
<td>Centralized</td>
<td>3.752</td>
<td>266.556</td>
</tr>
</tbody>
</table>

Table 7.6. Logic utilization and distribution

<table>
<thead>
<tr>
<th>Scheduling</th>
<th>Slice flip-flops</th>
<th>4 input LUTs</th>
<th>Occupied slices</th>
<th>Gate count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributed</td>
<td>274</td>
<td>1094</td>
<td>726</td>
<td>9639</td>
</tr>
<tr>
<td>Centralized</td>
<td>278</td>
<td>1086</td>
<td>772</td>
<td>9209</td>
</tr>
</tbody>
</table>
7.3 Ludesi

Ludesi is an image analysis algorithm used to analyze 2D protein gels. This algorithm is not that heavy on the type of program constructs that were displayed in the Permutation application in Section 7.1, therefore the benefit of using a centralized scheduler is limited. The decrease in number of registers used to implement the design stays at 2.7 percent. Figure 7.5 shows the Ludesi application before and after introduction of the central scheduler. The graph to the left in the figure shows the application using the distributed scheduler, and the graph on the right shows the application using the centralized scheduler. The figures does not display any major changes, which is reflected in the small change in number of registers used to implement the design.

<table>
<thead>
<tr>
<th>Scheduling</th>
<th>Flip-Flops</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributed</td>
<td>8311</td>
<td></td>
</tr>
<tr>
<td>Centralized</td>
<td>8089</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.8. Timing summary

<table>
<thead>
<tr>
<th>Scheduling</th>
<th>Min period [ns]</th>
<th>Max frequency [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributed</td>
<td>8.470</td>
<td>118.062</td>
</tr>
<tr>
<td>Centralized</td>
<td>8.251</td>
<td>121.202</td>
</tr>
</tbody>
</table>

Table 7.9. Logic utilization and distribution

<table>
<thead>
<tr>
<th>Scheduling</th>
<th>Slice flip-flops</th>
<th>4 input LUTs</th>
<th>Occupied slices</th>
<th>Gate count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributed</td>
<td>8149</td>
<td>9390</td>
<td>7493</td>
<td>1,101,817</td>
</tr>
<tr>
<td>Centralized</td>
<td>7927</td>
<td>9644</td>
<td>8598</td>
<td>1,101,100</td>
</tr>
</tbody>
</table>
Figure 7.5. Data dependency graph for the Ludesi application, without central scheduling on the left and with central scheduling on the right.
7.4 BLAST

Basic Local Alignment Search Tool also known as BLAST is an application widely used in bioinformatics. It is used to compare biological sequences. It can for example be used to compare sequences in the human genome with sequences from an animal to see if there are similarities. To conduct a search one needs a query sequence and a sequence to search against, for example in a database. The algorithm searches the database for sequences that match the query sequence. The sequences found are scored depending on how good a match they are. The algorithm basically consists of a number of filters. Since the amount of data that the algorithm searches through is very large the algorithm is constructed so that the first filtering step removes a lot of the sequences that are not interesting, this is done with a Bloom filter and false positives are possible, but false negatives are not. A Bloom filter is probabilistic, [2] and can not guarantee optimal alignment, speed is preferred over accuracy. The next step is widening the search and looking at the characters around the subsequences found and comparing them to the search query. The final step consists of the Smith Waterman algorithm which is guaranteed to find the optimal local alignment. [3] Figure 7.6 shows the BLAST application using the distributed scheduler on the left and shows the algorithm after implementation of a centralized scheduler on the right. A number of big folding fans can be seen in the figure, they are the same type of constructs found in the Permutation application, large copy and split nodes. This is the type of constructs that benefit from using a central scheduler. This design displays a decrease in number of registers with about 21 percent.

<table>
<thead>
<tr>
<th>Scheduling</th>
<th>Flip-Flops</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributed</td>
<td>45380</td>
</tr>
<tr>
<td>Centralized</td>
<td>35925</td>
</tr>
</tbody>
</table>

Table 7.10. Register report

<table>
<thead>
<tr>
<th>Scheduling</th>
<th>Min period [ns]</th>
<th>Max frequency [MHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributed</td>
<td>9.188</td>
<td>108.843</td>
</tr>
<tr>
<td>Centralized</td>
<td>22.690</td>
<td>44.073</td>
</tr>
</tbody>
</table>

Table 7.11. Timing summary
Figure 7.6. Data dependency graph for the Blast application. With distributed scheduling on the left and centralized scheduling on the right.
Table 7.12. Logic utilization and distribution

<table>
<thead>
<tr>
<th>Scheduling</th>
<th>Slice flip-flops</th>
<th>4 input LUTs</th>
<th>Occupied slices</th>
<th>Gate count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributed</td>
<td>44463</td>
<td>45953</td>
<td>38111</td>
<td>13,456,303</td>
</tr>
<tr>
<td>Centralized</td>
<td>35019</td>
<td>38077</td>
<td>30585</td>
<td>13,309,056</td>
</tr>
</tbody>
</table>
Chapter 8

Results

The results of the benchmark applications will be compared and discussed in this chapter to evaluate the centralized scheduler. The benchmark applications are presented in Chapter 7. The benchmark applications show that the centralized scheduler results in a smaller design for certain types of algorithms, those are the types of algorithms that contain large amounts of copy and split/make nodes as illustrated by the Permutation application. The Permutation application shows a 99 percent decrease in number of registers. This is not a real application, but an application created to illustrate the program constructs that will benefit from using a centralized scheduler, hence the dramatic decrease. The Ludesi and Shift application lacks the type of program constructs present in the Permutation application and consequently show a 2.7 percent decrease in number of registers used to implement the design for the Ludesi application and 0 percent decrease for the Shift application. The largest application used to evaluate the centralized scheduler in this thesis is the BLAST application, it has the type of constructs found in the Permutation application, large copy and split nodes, and would therefore be a good candidate for centralized scheduling. The synthesis reports confirm that, it shows a 21 percent decrease in number of registers used to implement the design.

The evaluation of the centralized scheduler have shown that the process of place and route becomes more difficult when using the centralized scheduler. This to an extent that the PAR tools fails to conform to certain timing requirements when synthesizing the BLAST application. The reasons for this might be that it is simply to difficult for the PAR tool when the size of the design is getting larger. The fill grade of the FPGA with BLAST loaded can be seen in Figure 5.1, the red line in the same figure indicates a timing path that is too long. The path stretches across the whole chip to get to a certain memory. This might be solved by moving the memories, but the memories in question are hard-wired on the FPGA and can not be moved. Another reason for the PAR to fail might be that the transformations and optimizations has removed certain nodes and connections that might not be necessary for the function of the application but it might give the PAR tool some extra leeway.
Bibliography


Appendix A

Hardware Generation

The hardware generated by the compiler comes in the form of VHDL code. VHDL code for the three new nodes, NMVC, M2Z, Z2M, must be added. The VHDL code in this chapter displays the nodes functions.

A.1 M2Z

As discussed in Chapter 4.3 regarding simulation models, the M2Z node copies the data from input0 to output0 while outputting a std logic vector of length 1 with value 1 on output1. This can be seen in the VHDL code below.

```java
public void dumpVHDLEntity(PrintWriter f)
{
    dumpVHDLEntityHeader(f);
    f.println("architecture RTL of " + VHDLname() + " is");
    f.println("begin");
    f.println("Dout0 <= Din0;");
    f.println("Dout1 <= "1");
    f.println("end RTL;");
}
```

A.2 Z2M

The Z2M node copies the data from input0 to output0. The data at input1 is not used, input1 in a control signal used by the scheduler.

```java
public void dumpVHDLEntity(PrintWriter f)
{
    dumpVHDLEntityHeader(f);
    f.println("architecture RTL of " + VHDLname() + " is");
    f.println("begin");
    f.println("Dout0 <= Din0;");
    f.println("end RTL;");
}
```
The NMVC node generates the value true on every output. This can be seen in the following VHDL code.

```java
public void dumpVHDLEntity(PrintWriter f)
{
    dumpVHDLEntityHeader(f);
    f.println('architecture RTL of ' + VHDLname() + ' is ');
    f.println('begin ');
    for (int i = 0; i < outputs; i++)
    {
        f.println(' Dout' + i + ' <= ' + '1' + ';');
    }
    f.println('end RTL; ');
}
```