Master of Science Thesis in Electrical Engineering
Department of Electrical Engineering, Linköping University, 2018

Accelerated Simulation of Modelica Models Using an FPGA-Based Approach

Alexander Yngve, Herman Lundkvist

Master of Science Thesis in Electrical Engineering

Accelerated Simulation of Modelica Models Using an FPGA-Based Approach

Alexander Yngve, Herman Lundkvist

LiTH-ISY-EX--17/5106--SE

Supervisor: **Mario Garrido**
ISY, Linköpings Universitet
Mattias Kling
Saab
Per Holmbom
Saab

Examiner: **Oscar Gustafsson**
ISY, Linköpings Universitet

Division of Computer Engineering
Department of Electrical Engineering
Linköping University
SE-581 83 Linköping, Sweden

Abstract

This thesis presents *Monza*, a system for accelerating the simulation of models of physical systems described by ordinary differential equations, using a general purpose computer with a PCIe FPGA expansion card. The system allows both automatic generation of an FPGA implementation from a model described in the Modelica programming language, and simulation of said system.

Monza accomplishes this by using a customizable hardware architecture for the FPGA, consisting of a variable number of simple *processing elements*. A custom compiler, also developed in this thesis, tailors and programs the architecture to run a specific model of a physical system.

Testing was done on two test models, a water tank system and a Weibel-lung, with up to several thousand state variables. The resulting system is several times faster for smaller models and somewhat slower for larger models compared to a CPU. The conclusion is that the developed hardware architecture and software toolchain is a feasible way of accelerating model execution, but more work is needed to ensure faster execution at all times.

Acknowledgments

Thanks to Mario Garrido and Oscar Gustafsson at LiU and to Mattias Kling and Per Holmbom, Per Nikolaisen at Saab for your guidance and feedback. Also thanks to Magnus Ingmarsson for your constructive critisism on the structure and language of the report. Lastly, thanks to Saab for allowing us to work on this interesting project, and for buying expensive hardware for us to play with!

Linköping, november 2017
Alexander Yngve and Herman Lundkvist

Contents

1

Introduction

During the last decade there has been a strong push towards model-based systems engineering (MBSE) for the simulation of physical systems. In the MBSE design methodology domain-specific models are used as the main way of communication, verification and design space exploration [1].

The models are often made using a modeling environment such as Dymola or OpenModelica, and can represent any aspect of a physical system, such as

1. the hydraulic system of an airplane,
2. the wheels and tires of a car,
3. individual actors in a world simulation.

Depending on the application, the modeling of physical systems can be done in several ways, but a popular method is to use a programming language specifically developed for this purpose. Two examples of such programming languages are Modelica, which this thesis will focus on, and Simscape.

1.1 Motivation

The models are possibly connected to a larger system and simulated to verify their functionality before being implemented as a real system. In theory, using models for design space exploration by running simulations as fast as possible allows engineers to quickly iterate different solutions. When using the models for verification, the performance is also critical since you may want to test a large number of cases or you might want to test the system with real-time constraints.

However, the reality of model simulation is often very different, with models far to slow for real-time use or large scale testing. The models are represented as different forms of differential equations, such as ordinary differential equations

(ODEs) or differential algebraic equations (DAEs). In general, these equation systems cannot be solved analytically and may require a large amount of numerical calculations, which means slow simulations at a fraction of the desired real-time performance [1].

One way of improving the performance is by using some form of hardware acceleration. In recent years many computationally heavy tasks, such as video decoding and other signal processing, have been accelerated by graphics processing units (GPU) [2]. Another form of acceleration which is expected to become more prevalent in the future is the use of field programmable gate arrays (FPGAs), which can be used as a form of programmable hardware. As a platform for accelerating applications, FPGAs fill the gap between the more general GPU, and the fixed function, application specific integrated circuits (ASICs). They have the possibility of performing certain applications faster than a GPU by providing a dedicated hardware architecture, while requiring lower development cost than ASICs.

However, compared to GPUs, the use of FPGAs in the form of an accelerator card in conjunction with a host computer is not as extensively explored. The possibility of increased performance, makes investigation of porting performance critical applications to FPGAs an interesting undertaking.

In fact, FPGAs may be more suitable than GPUs for certain types of physical models. As stated in a thesis by Stavåker [3], not all physical models are inherrently data-parallel, which is the type of problems GPUs excel at. The greater flexibility of FPGAs may provide means to mitigate this problem.

Faster model simulations, for example with the help of FPGAs, would in turn lead to faster design iterations and more confidence in the system under test with the ability to run more tests.

Writing an ad-hoc FPGA implementation for every model to be simulated, however, would likely mean that the benefits of decreased simulation time be outweighed by the increased development costs. A better solution would instead be to devise a system that could automatically generate an FPGA implementation using existing modeling tools, which is what this thesis explores.

1.2 Aim

The aim of this master thesis is to contribute to the knowledge of running model simulations on FPGAs, specifically by solving ODE systems. To achieve this, two principle goals have been outlined.

The first goal is to develop a system that takes a complete model written in Modelica as input, and as output produces an implementation which can simulate that model on an FPGA. In order to accomplish this, multiple hardware architectures will be considered at the design phase, and a motivation for the chosen one will be given.

The second goal is to evaluate the developed system and to compare simulation using the system against an existing program that allows simulation on a general-purpose CPU. For the latter program, this thesis will use the OpenMod-

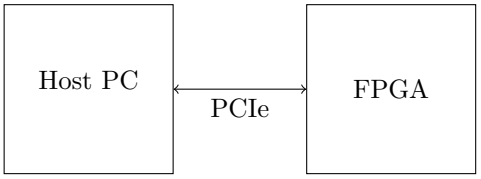
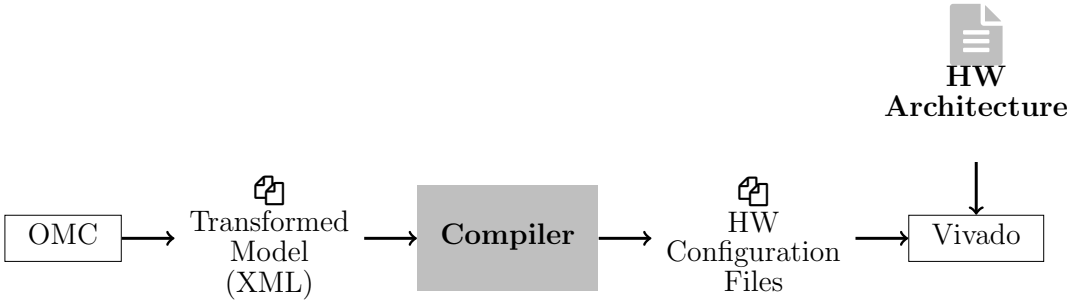
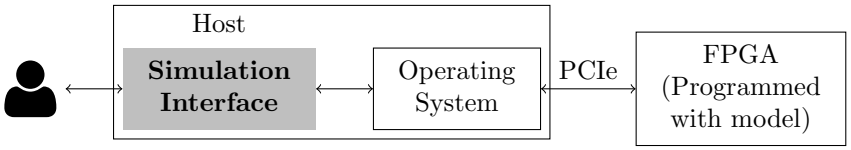


Figure 1.1: *Overview of the hardware platform.*



(a) *Programming phase.*



(b) *Simulation phase.*

Figure 1.2: *The initial design of the components of the proposed system and how they interact. The gray parts were developed in this thesis.*

elica Compiler (OMC).

1.3 Inital System Design

Before development began, an initial, high-level system design, pictured in Figures 1.1 and 1.2, was produced. This was used to limit the scope of the thesis and to provide a context for the outlining of the evaluation, including the formulation of the research questions. The high-level design was performed after intial design exploration, and was largely adhered to in the final implementation.

Figure 1.1 shows the hardware platform that was chosen: a host computer that uses PCI express (PCIe) to communicate with an FPGA card.

Figure 1.2 shows the high-level components in the initial design. It was decided that the usage of the system be split into two distinct phases:

1. A programming phase that takes a model description as input and as output

produces a bitstream that can be used to simulate that model on the FPGA.

2. A simulation phase where a user can interact with the FPGA that has been programmed with a model from a previous programming phase.

The reason for this divison was that a model would typically be programmed once, but simulated multiple times.

In the programming phase, the system would use OMC to handle syntactic and semantic processing. Note this only a subset of the functionality of OMC, unlike on a CPU where it is also used to perform the actual simulation (this is explained in more detailed in Chapter 2). The reason for using OMC for this part was to reduce development costs.

To reduce the design space, the mechanism for generating FPGA implementations from model descriptions was split into two parts: first, a general hardware architecture having some customizability; second, a complier capable of tailoring and programing the hardware architecture to simulate the specific model described in the OMC output.

The bitstream would be produced by using hardware configuration files – including memory contents and or hardware description language (HDL) code – from the compiler and static HDL code describing the hardware architechture as input to the Vivado synthesizer.

In the simulation phase, the system would use an interface developed in this thesis allowing a user to: manipulate model parameters, start a simulation, and to read the simulation results. The simulation interface would use the operating system (OS) of the host computer to communicate with the FPGA.

1.4 Research Questions

1. How does the proposed system compare to simulation using OMC in terms of:

   (a) The time to complete a simulation, and

   (b) correctness.

2. How fast can data be transferred to and from the FPGA card?

3. What is the limiting factor in FPGA resource utilization of the proposed system?

4. Are there any bottlenecks during execution on the FPGA? If so what are they?

The definition and measurement of the time to complete a simulation, the correctness of a simulation, and efficiency of the system is explained in chapter 4.

1.5 Delimitations

Since it is a big undertaking to solve general DAE systems, this project only focuses on models with ODE systems; the general case of simulating all possible models on an FPGA is not covered but instead encouraged as future work. For the same reason, the project only considers the explicit fixed step solver *Euler forward*.

In addition the developed system only accepts a subset of the Modelica language, it only handles equations without discontinuities and thus neither event handling nor control flow statements.

Also, the systems analyzed in this thesis are ODEs with multivariate polynomials without algebraic loops. This excludes systems with non-linear functions such as trigonometric functions and square root.

It would also have been interesting to do benchmarking compairsons with GPU and parrallel CPU implementations. However, this was not done due to time constraints.

1.6 Background

This thesis is the result of a project commissioned by Saab, a Swedish defense and security company developing fighter aircrafts. During the design and verification, the company uses models and simulations are extensively. To this end, they are interested in new hardware platforms to use in the simulators, one of which being FPGAs.

2

Theory

This chapter contains some important theory regarding modeling, differential equations and previous work within the area of solving modeling problems with hardware acceleration.

2.1 Differential Equations

A fundamental mathematical concept that commonly arises in many engineering fields, including the simulation of physical systems, are differential equations [3, p. 12][4, p. 4-5].

Differential equations are equations where the unknowns can be functions and derivatives of functions. If there is only a single unknown variable, the differential equation is called an *ordinary differential equation*. If there are two or more unknown variables it is a *partial differential equation*.

The differential equation is also characterized by *order*. The order of the differential equation is equal to the highest order derivative within it [5]. An n:th order differential equation can always be converted to an n-dimensional system of first order equations by introducing more variables.

Most differential equations which arise in real life cannot be solved analytically or by hand. Instead, a numerical method for finding an approximation is needed [6]. This section will focus on ODEs. Unless otherwise stated, all information in Section 2.1 is sourced from [6].

2.1.1 Ordinary Differential Equations

Systems of first order ordinary differential equations in general have the following structure

$$\mathbf{y}' = \mathbf{f}(t, \mathbf{y}) \tag{2.1}$$

where $\mathbf{y}$ and $\mathbf{f}$ are vectors of equal length, t is time and $\mathbf{y} = \mathbf{y}(t)$. $\mathbf{f}$ can be a nonlinear function.

2.1.1.1 Initial Value Problems

Initial value problems (IVPs), are problems where the initial state of the system is known, which means $\mathbf{y}(0) = \mathbf{c}$ where the vector $\mathbf{c}$ is given. The initial state can be used to approximate the system's state in the future with methods described in Section 2.1.2.

2.1.2 Numerical Methods

Since most differential equations cannot be solved analytically, numerical methods are used instead. Most numerical methods work by having a known point on the solution curve and in different ways approximating a direction towards the next point to take a step in, then start a new iteration with the same process. The distance needed between the points on the curve is a trade-off between performance and accuracy and is one of the main differing aspects of the many numerical methods. The number of calculations in each iteration is another difference.

The forward Euler method is arguably the simplest numerical method for solving ODE systems. It calculates the next point on the solution curve by taking a step of fixed size from the previous point in the direction of the derivative. This is illustrated in Figure 2.1: 2.1a shows a single iteration step, while 2.1b shows multiple iterations with intermediate values A_0-A_4 compared to an exact analytical solution.

For an IVP of the same type as equation 2.1, the method is applied with the iteration formula in equation 2.2.

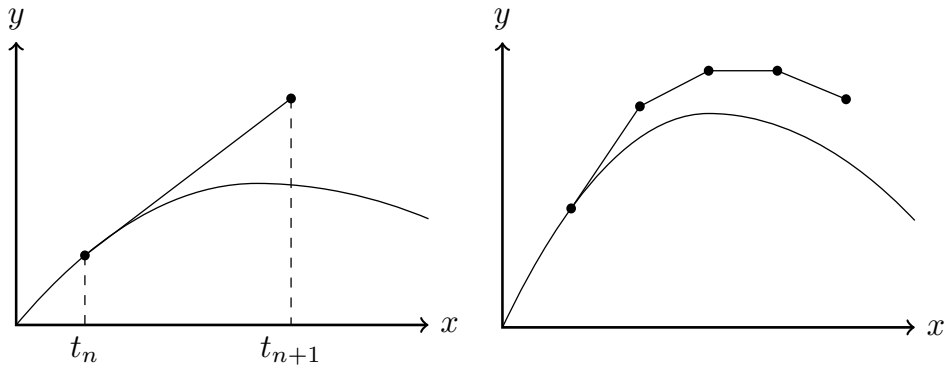
$$\mathbf{y}_n = \mathbf{y}_{n-1} + h\mathbf{f}(t_{n-1}, \mathbf{y}_{n-1}) \tag{2.2}$$

where h is the step size. A smaller step size leads to increased accuracy at the expense of more iterations, thus lower computation speed.

A related method to forward Euler is backward Euler. Its formula, equation 2.3, looks very similar to forward Euler with the exception of the arguments to $\mathbf{f}$. The direction to take a step in is now approximated by the future point $(t_n, \mathbf{y}_n)$!

$$\mathbf{y}_n = \mathbf{y}_{n-1} + h\mathbf{f}(t_n, \mathbf{y}_n) \tag{2.3}$$

Methods where $\mathbf{y}_n$ appears on both sides of the iteration formula, such as backward Euler, are called *implicit* methods. When $\mathbf{y}_n$ is calculated directly from



(a) *A single time step is taken in the direction of the derivative.*

(b) *Multiple time steps. The bottom curve is the analytical solution.*

Figure 2.1: *Illustrations of the forward Euler method.*

already known values, such as in forward Euler, the method is called *explicit*. Implicit methods generally require solving a set of nonlinear equations at each time step, for example via Newton iteration, which makes each iteration more computationally demanding. Explicit methods are simpler and each iteration can be calculated in a straightforward manner.

The benefit of backward Euler (implicit methods) are their increased *stability* and performance for *stiff* problems, but this kind of problems are outside the scope of this report.

2.2 Modeling

As explained in the introduction, Modelica is a programming language that can be used to describe models of physical systems.

Modelica is an equation based language with some features from ordinary computer languages such as object orientation and regular flow control. The actual modeling work is mostly done in a graphical integrated development environment, such as OpenModelica or Dymola, where components, such as hydraulic cylinders or tanks, are connected via ports in a diagram. The components can in turn be described by diagrams with connected subcomponents or on the bottom level by Modelica code.

In the end, the models can be transformed to a set of equations – specifically differential, algebraic and discrete equations – which can be used to simulate the models [7].

2.3 Simulation of Physical Systems

2.3.1 State Variables

To understand simulation of physical systems, it is important to know the concepts *state variables* and *state-determined systems*. This is because many physical systems can be defined in terms of these. Such systems fulfill the following requirements:

State-determined systems and state variables In order for a system to be state-determined, it must be possible to obtain the future values of all variables in the system using:

- the future input of to the system, and
- the values of a subset of the system's variables at some initial time – these are called state variables [4, p. 9].

The state variables typically appear as derivatives in the differential equations.

2.3.2 Causalizing Equation Systems

Relations and laws that govern physical systems are often described in a form that is *acausal*, meaning that the equals sign in an equation denotes equality rather than assignment [8, p. 255]; and *implicit*, which means that an equation is not solved for a specific variable, i.e. not written on the form $y = g(\mathbf{x})$ [5]. It stands to reason that modeling physical system becomes easier if the modeling environment allows equations to be expressed in the aforementioned way.

However, this poses a problem for simulation because a digital computer must evaluate the system as a sequence of operations. Tools like Dymola and OpenModelica solve this problem by transforming and sorting the equations to establish:

- *Vertical* order - which equation to be used to solve for each variable.
- *Horizontal* order - in what order the variables should be solved [8, p. 4].

Solving a variable for an equation often involves the use of numerical solvers, since Modelica can express problems that cannot be solved analytically in the general case. This is typically true for differential and non-linear equations.

There are multiple methods that can be used to achieve a system that can be simulated, and one important algorithm to this end is Tarjan's algorithm [8, p. 256]. Given a system of implicit DAE equations, the algorithm attempts to sort the equations vertically and horizontally. If completely successful, the sorting results in a fully explicit system. Otherwise the sorting may give rise to so called *algebraic loops*. These are sets of equations that are tightly coupled, and thus need to be solved together. Depending on whether the problem is linear or non-linear, a linear solver or Newton's method might respectively be used to resolve the algebraic loops. In addition, Tarjan's algorithm can also be used for detecting higher index problems. Solving higher index problems is a complex subject, and

is out of scope for this thesis. Nonetheless, two approaches for solving them are: the use of index reduction methods, and using solvers for higher index DAEs.

In this thesis, causalization will be handled by OMC, and as such the steps of Tarjan's algorithm will not be given here. For a detailed explanation please refer to [8, p. 256].

2.3.3 Updating the state variables

A successful causalization of the equations results in a series of expressions that can be executed by a computer to obtain the values for all variables and derivatives. However, to update the state variables, numerical solvers are employed. As explained in Section 2.1.2, these work by approximating the value of the state-variable in the next iteration based on the derivatives of current and previous iterations.

The simulation can thus be broken down into two principal parts.

- Calculation of intermediate variables and derivatives using the state variables.
- Updating the state variables using the numerical solver.

2.3.4 Task Graphs

The methods described above enables simulation of physical models. But to speed up the simulation without simply increasing the serial execution speed of the computer platform, parallelism has to be exploited. One way of identifying parallelism in the model simulation is to analyze the data dependencies between the expressions.

A task graph is a useful tool for illustrating the data dependencies of a sequence of operations. It can be represented as a directed acyclic graph (DAG) where every node represents a set of operations that must be performed, and an edge from a node a to a node b, means that the operations of a must be performed before those of b.

The number of operations in a node, also called a *task*, may vary depending on use-case. In the following section a task is a single equation, i.e. an expression that is saved to a variable.

2.3.5 Simulating an RLC filter

To illustrate the process of modeling and simulating a physical system, an RLC filter will be studied as an example. The simulation will use Euler forward as a numerical solver. The electrical schematic of the filter is shown in Figure 2.2.

2.3.5.1 Modeling

First, variables and equations have to be identified to describe the system. To simulate both the current and the voltage, the values for eight variables need to

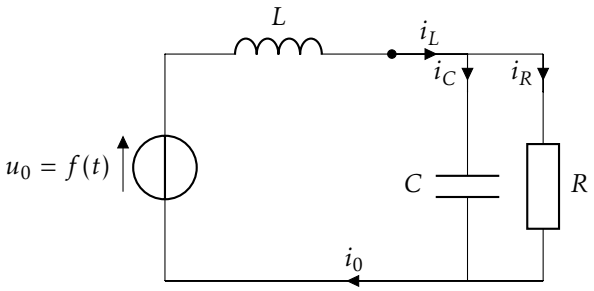


Figure 2.2: *An RLC low pass filter.*

be calculated since there are four components. With eight unknowns, the same number of equations are needed.

The constitutive equations gives one equation for each component.

$$u_0 = f(t) \tag{2.4}$$
$$u_L = L di_L/dt \tag{2.5}$$
$$i_C = C du_C/dt \tag{2.6}$$
$$u_R = R i_R \tag{2.7}$$

Two more equations can be derived using Kirchoff's voltage law:

$$u_0 = u_L + u_C \tag{2.8}$$
$$u_0 = u_L + u_R \tag{2.9}$$
$$\tag{2.10}$$

and another two can be derived using Kirchoff's current law

$$i_L = i_R + i_C \tag{2.11}$$
$$i_0 = i_L \tag{2.12}$$

Note that equations 2.5 and 2.6 are ODEs, and are solved numerically in this example. Additionally, the equations contain i_L and u_C, which are the state variables of the system.

Since Euler forward is an explicit method, it means that the values for i_L and u_C for the current iteration are considered to be known (the initial values are used for the first iteration). Instead, the derivatives di_L/dt and di_L/dt are treated as unknowns since they are needed to compute the values of i_L and u_C for the next iteration.

2.3.5.2 Causalization

The eight equations mentioned above describe the system implicitly: they do not make it explicitly clear in what order operations should be performed to obtain values for each variable.

Using Tarjan's algorithm, one possible causalized version of the system is:

$$i_0 := i_L \tag{2.13}$$
$$u_0 := f(t) \tag{2.14}$$
$$u_L := u_0 - u_C \tag{2.15}$$
$$u_R := u_0 - u_L \tag{2.16}$$
$$di_L/dt := u_L/L \tag{2.17}$$
$$i_R := u_R/r \tag{2.18}$$
$$i_C := i_L - i_R \tag{2.19}$$
$$du_C/dt := i_C/C \tag{2.20}$$

Observe that the equals sign, denoting equality in equations (2.4)-(2.12), have turned into assignments. In addition to being causalized, the system does not contain any algebraic loops.

2.3.5.3 A Task Graph of the Causilized System

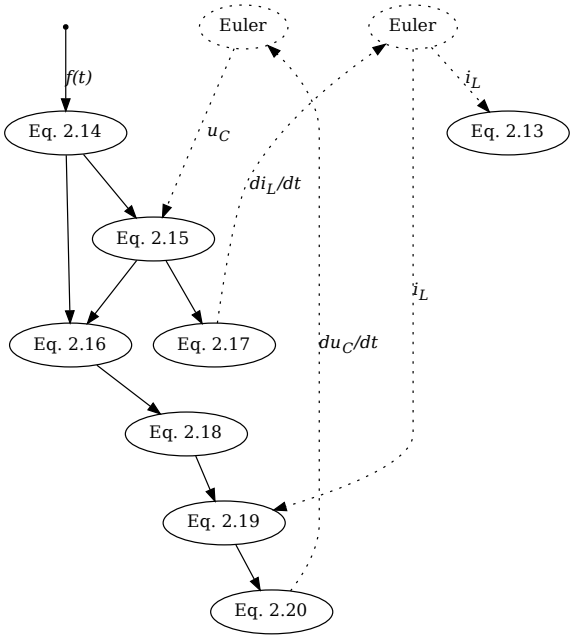


Figure 2.3: *A task graph for the filter circuit with Euler forward as the solver. The solid lines correspond to task results being used for computing variables, whereas the dotted lines show usage for computing state-variables.*

Figure 2.3 shows a task graph for the causalized equation system of the RLC circuit in the previous section, with round nodes corresponding to the assignments (2.13)-(2.20). The task graph also includes the operations performed by the Euler forward solver, and thus shows all operations that are performed during one time-step. The figure assumes that the system is initialized and that the values for the constants R, L, and C are included in the nodes where they are used.

As the task graph illustrates, simulation with explicit fixed-step methods can be divided into two distinct phases: updating the derivatives and updating the state-variables.

2.4 OpenModelica Compiler

The goal of the thesis is to provide a system that can generate synthesizable HDL code for a subset of the Modelica language. However, writing a Modelica compiler can be a laborious undertaking. Therefore the work of this thesis intends to be used with an existing compiler, the OpenModelica Compiler (OMC), thereby allowing re-use of front-end components that handle syntactic and semantic analysis.

OMC is a part of OpenModelica, which is a collection of tools and environments for developing and simulating Modelica models. The Compilation of modelica models into programs that can be simulated can be broken down into five steps [9] (see Figure 2.4):

Model Flattening First, a number of Modelica files are given as input, syntactic and semantic analysis is performed, and a so-called flat model is produced. This model is an elaborated version of the original source code, where all object-oriented features have been removed.

Equation Sorting Then, the equations in the flat model are sorted vertically and horizontally, see Section 2.3.2.

Optimization The sorted equations are analyzed and optimized with the aim of simplifying them and/or improving simulation speed.

Code Generation A code generator takes a representation of the model to be simulated, including the sorted and optimized equations, and generates code that can perform the simulation. The default target language is C.

Compilation of Simulation Code Finally, the simulation code is compiled with a compiler for the target language. The resulting executable can be run to perform the actual simulation.

For the code generation step, OMC has an option to generate an XML-file containing the sorted equations with the mathematical expressions structured into trees, instead of generating C code. The system developed in this thesis will use these XML files as the basis for generating the HDL-code.

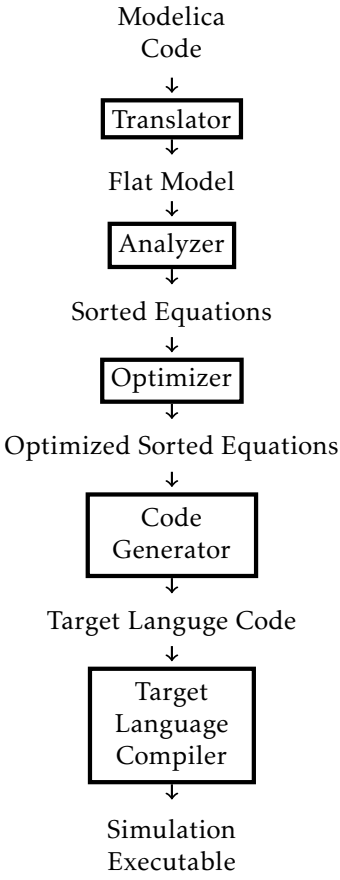


Figure 2.4: *The compilation phases of OMC.*

Another way of structuring the final system could have been to implement the code-generation as part of OMC. However, there were two reasons why this was not done. First, the authors would have had to learn MetaModelica since this is the principal language used for code generation, which would not be possible within the time frame of this work. Second, keeping the code generation separate from OMC would most likely simplify integration with other Modelica compilers.

2.5 Field Programmable Gate Arrays

Field programmable gate arrays, FPGAs, are chips which can be reconfigured. Reconfigurable computing devices fill the gap between fixed-function ASICs and software running on CPUs. FPGAs allows higher performance than software

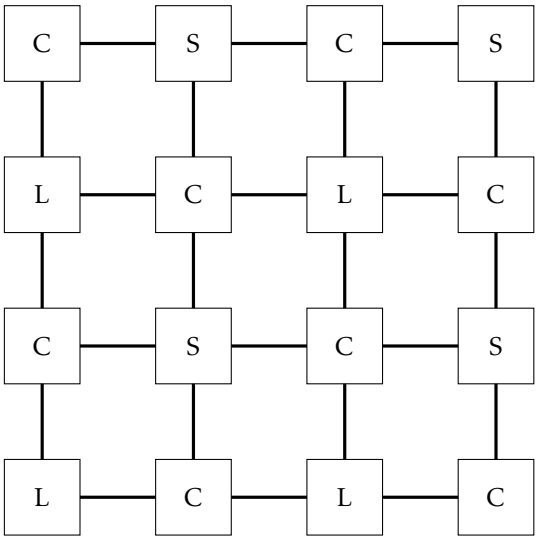


Figure 2.5: *FPGA block diagram.*

while having greater flexibility than an ASIC [10].

2.5.1 FPGA Internals

FPGAs are often constructed as a large 2D-grid of *logic*, *connection* and *switch* blocks. This is illustrated in Figure 2.5, where L is a logic block, C is a connection block and S is a switch block.

2.5.1.1 Logic Blocks

The logic blocks of an FPGA are components which actually perform computations. The logic blocks are often implemented as an n-input look-up table with a bypassable D flip-flop, which can perform any boolean operation on n inputs and optionally provide 1 bit of storage. n is often a small number, such as 3 or 4. Another way of implementing a logic block is to use a more advanced computational element, such as an adder, multiplier or an even larger DSP (digital signal processing) block. Logic blocks can also contain memory elements, which gives the engineer easy access to (volatile) storage. When not used as storage, they can function as large LUTs [10].

2.5.1.2 Routing Blocks

In order to calculate arbitrary logic functions, multiple logic blocks must be connected together. This requires a flexible routing network. The connection blocks contain programmable multiplexers selecting which logic block terminals are connected to which routing channel. At switch blocks the routing channels can change direction from horizontal to vertical and the other way around [10].

2.5.2 System Design

A heterogeneous system normally contain all the parts of a standard computer, CPU, memory, peripheral devices and one or more accelerators with different architecture than the main CPU(s). The accelerator in this case is an FPGA. The FPGA can be integrated into a larger system in different ways. The simplest way is to treat it as a peripheral device. This leads to a higher cost of communication, but may still be sufficient if the application is computationally bound.

Another approach is to move the FPGA closer to the CPU, enabling very tight coupling and fast communication. The FPGA can be placed on the same chip as the CPU and be treated as a reconfigurable accelerator. The other way around is also possible, placing a hard CPU core in the reconfigurable fabric [11]. The latter is already implemented in products from both Xilinx [12] and Intel [13].

2.5.3 Development Process

The development process and tooling of FPGAs differs from a pure software development flow. It involves the following stages:

1. Hardware design
2. Specification
3. Simulation
4. Mapping
5. Placement
6. Routing
7. Testing and Verification

The *specification* of the hardware circuit is often done in a hardware description language such as VHDL. The next three stages are performed by a vendor specific toolchain. The *mapping* stage takes the VHDL code and produces a gate-level description of the circuit to be generated. The *placement* process places the mapped gates on specific logic blocks in the FPGA, thereby generating part of the bitstream. *Routing* then finds the necessary communication channels between the logic blocks and generates the bitstream, the configuration bits for the connect and switch blocks. These three stages are equivalent to compilation of code in a programming language [10].

Writing HDL code consist of describing the structure of the hardware. This can be a daunting task for application designers who wish to focus on describing an algorithm for solving a problem, not specify the hardware to run it. To this end, several efforts for generating hardware circuits from behavioural descriptions have been made. *High-level synthesis* is when a hardware circuit is generated from a software programming language such as C [11]. Recently, there have also been progress towards using OpenCL for programming FPGAs. This means code which describes an algorithm could be written once and deployed on a multitude of computing devices, FPGAs, GPUs or CPUs [14].

2.6 Previous Work

Previous work within this area involves efforts to parallelize equation-based models in general as can be read in Section 2.6.1 and work on solving differential equation systems on FPGAs as can be read in Section 2.6.2.

2.6.1 Parallelization of Equation-Based Models

The thesis by Stavåker [3], summarizes different ways to parallelize execution of Modelica models on a GPU. He mentions three approaches for utilizing parallel computations

1. Explicit parallel language constructs
2. Coarse-grained explicit parallelization
3. Automatic (fine-grained) parallelization

The latter has the advantage that it requires no intervention from the modeller. He further mentions three ways of achieving automatic parallelization

1. Parallelism over the method - the numerical solver is parallelized
2. Parallelism over time - multiple time steps are run at once
3. Parallelism over the system - multiple equations are solved at once

Methods for parallelization over the system and over the method by means of a task graph is then presented.

The results of the task graph parallelization are mixed, it is stated that it may have performance benefits when the equations have little dependencies between each other, otherwise the memory transfers in the system becomes a bottleneck. This stems from the fact that solving an ODE system is not inherently data-parallel, which is the kind of tasks GPUs excel at. Further work on automatic parallelization, specifically the clustering and scheduling of the task graph is made in [15].

In addition, there is also the possibility of combining the different ways when parallelizing the solution. This is the case in [16], where methods for parallelizing embedded Runge-Kutta methods over both the method and the system on multi-core and distributed systems were considered. The authors demonstrated methods that achieved speed-ups by improving the data locality and reducing synchronization. In addition, methods that specifically targeted models where the equations depended on a few, neighboring equations, which is often the case when discretizing problems, achieved significant speed-ups. However, the improvements in the article were primarily aimed at resolving overhead related to cache-misses and synchronization. Because of the likelihood of having a custom communication network, and a much simpler memory hierarchy, it is not immediately clear how the benefits could be utilized in an FPGA implementation.

2.6.2 Executing Models on FPGAs

2.6.2.1 Using Processing Elements

There are already a few published methods for implementing custom hardware architectures for accelerating models of ODE systems. Successful examples can be found in a series of related papers [17], [18], [19] and [20] that investigate different techniques to simulate linear ODE systems of physical models on an FPGA using fixed-step solvers. In essence, the proposed method is to use a number of processing elements (PEs), i.e. simple computational nodes that calculate the state variables for one or more ODEs, and connect these according to the data dependencies of the equation system.

There are multiple advantages of this method. Firstly, the problem is parallelized to a large extent, since each PE can update its state variables independently of the others. Secondly, communication overhead can be reduced since the interconnect is point-to-point between the PEs and tailored towards each model.

The authors develop and compare PEs with different data paths including fixed-function versions used for computing specific equations, and more general versions.

Furthermore, the authors experiment with ways of combining different types of PEs. They show that using a heterogeneous network of PEs, i.e. containing both general and fixed-function PEs can yield substantial speed-up compared to using HLS on FPGA, and executing on CPU and GPU platforms.

In fact, for the five models that were tested (physiology models, with different characteristics, consisting of thousands of equations) the heterogeneous network was 9x-14x times faster than HLS [19, p. 222], 36x-60x faster than a single thread on a Intel I7-950 CPU, and 13.7X-29X faster than a NV GTX460 GPU [19, p. 223].

It is worth noting that the authors used fixed-point instead of floating-point computation. While being more efficient both area- and latency-wise the former is not as exact as the latter.

In [20], the execution time was further decreased due to increases in clock frequencies which were enabled by improvements in placement and routing. A process was developed that could be used to automatically improve the routing. Additionally, two methods for improving the placement were presented: a technique using embedded H-trees for models having a binary-tree-like structure, and a simulated annealing algorithms for general models.

2.6.2.2 Using Co-Simulation

Another approach for hardware acceleration of ODE systems can be found in [21]. This work focuses on creating an automated framework for speeding up the simulation of biomedical systems with a large number of cells. Instead of implementing the entire model in hardware, the authors opt for a co-simulation method by moving a computationally intensive part of the OpenCMISS simulator onto an FPGA.The proposed target platform is a general purpose PC with an FPGA accelerator card.

The simulation workflow has two important tasks: an ODE system is solved for each cell, and a spatial solver combines the result of these to provide the final answer. Arguing that the spatial solver lacks parallelism, the authors suggest calculating only the cell simulation on the FPGA, specifically by the use of a FPGA accelerator card PCIe connected to a host PC.

The hardware implementation does not use an intermediate hardware architecture layer, like processing elements. Instead, the mathematical operations of an explicit ODE system, combined with those of the Euler forward method, are arranged in a dependency graph and more or less directly mapped to the FPGA.This results in a pipeline on the FPGA where execution is broken down into multiple stages with registers holding intermediary results. A binding and scheduling algorithm is used to meet data and temporal dependencies according to the dependency graph. The implementation uses floating point cores to implement the mathematical operations.

The CPU-FPGA system was evaluated against multi-core CPUs with and without SIMD (Intel SSE) instructions and GPU implementations. For the three models that were tested, all having around 200,000 cells, speed-ups of 7.99x-32.15x compared to a single threaded CPU application were achieved. Although slower compared to a few setups, for instance a manually optimized GPU implementation, The CPU-FPGA system required the least amount of power by a great margin.

In [22], FPGA resource usage and execution speed was improved upon as a result of applying different strategies including the running multiple cell simulations in parallel and testing different floating point cores.

2.7 Additional Theory

The theory in the following sections are relevant for understanding concepts of the system that the thesis resulted in. However the need for these subjects only makes sense in light of the result. As such it may be easier for the reader to refer back to these, while reading the result.

2.8 Linux Character Device Drivers

The Linux Device driver used in the developed system is one that exposes a so called *character device*. This means that the device has been abstracted to appear as a stream of bytes. A driver for such a device implements system calls that are normally used to manipulate files. However instead of operating on files, the system calls use a file node that has been associated with the hardware device in question. The system calls may, among others, include the POSIX functions `open`, `close`, `read`, `write` [23, p. 6, ch. 1].

Examples of hardware devices that can be viewed as character devices are: consoles and serial ports.

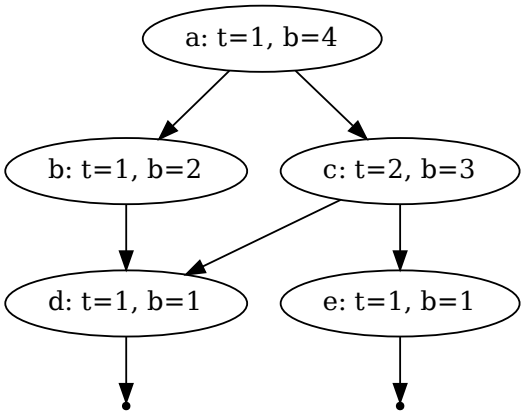


Figure 2.6: *A example of a DAG where t denotes the execution time, and b the b-level.*

2.9 Scheduling and Mapping

The definitions regarding static scheduling and mapping of DAGs are taken from [24].

The scheduling and mapping algorithm used in this thesis belongs to a class called *list scheduling*. Such algorithms keep a list of tasks that are ready for scheduling – i.e. have all dependencies satisfied – in the current state. They begin by assigning a priority to all tasks in the DAG, and inserting all entry tasks – tasks that have no dependent tasks – to the *ready list*.

The algorithms then proceed iteratively applying the following two rules:

1. Take a task with the highest priority from the ready list.
2. Assign it to a processor which allows the earliest execution time.

until all tasks have been scheduled.

There are different ways to assign priorities to the tasks, but the metric chosen in this thesis is the *b-level*. According to [24]: 'The b-level of a node n_i is the length of a longest path from n_i to an exit node.'

To illustrate how this definition is used to calculate b-levels, an example is provided in Figure 2.6. In it, the nodes *d* and *e*, each being adjacent to an exit node, have b-levels equal their execution time. For node *b* the only and longest path goes via node d, giving a b-level of 2. There are two longest paths for *c*, using either one gives the b-level. For node *a*, the b-level comes from a path via c.

3

Implementation

This chapter describes the system that the thesis resulted in at the hardware layer and the software layer. Section 3.1 contains a discussion of the considered hardware architecuteres and a motivation for selected one. Section 3.2 introduces the implemented system and Section 3.3 shows an example of the system, from input in the form of a Modelica file to the output of the bitstream. Section 3.4 and 3.5 describes the hardware- and software architecture in more detail.

3.1 Architecture Selection

Two different hardware architectures were considered during the design phase, each showing similarities to an architecture presented in Section 2.6.

3.1.1 Approach 1: Direct Mapping of Task Graph

The first approach considered was a, more or less, direct mapping of the task graph for a model into a custom hardware architecture. This meant using one functional hardware operator block for every operation, and connecting them according to the dependencies in the task graph.

The approach is similar to that described in Section 2.6.2.2. But whereas that work used the FPGA to compute a kernel of the problem, this work would use the FPGA to solve the entire model.

A problem that would need to be solved is related to timing. If one branch of a binary operation is slower than the other, the faster branch would need to be somehow stopped or delayed until the slower is complete. The timing could be solved asynchronously by using handshaking between branches, or synchronously by inserting extra delay registers in the faster branches.

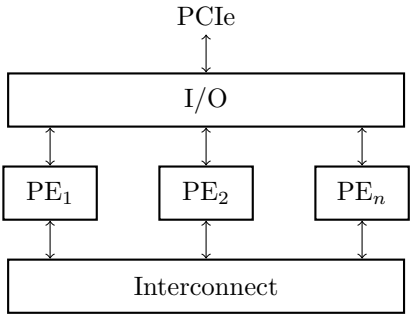


Figure 3.1: *Proposed architecture with processing elements.*

3.1.2 Approach 2: Connected Processing Elements

The other approach is similar to that referred to in Section 2.6.2.1, and uses an extremely simple processing element (PE) only supporting operations for arithmetic and storing. However the previous work assumed that the models could be divided into ODEs with few inter-dependencies that could be calculated independently, with only one communication and synchronization step per iteration. This made a point-to-point interconnect suitable.

In our suggestion, we instead allow for the flat-model generated by OMC to contain more fine-grained inter-dependencies between equations, with multiple synchronization points. To allow for more dependencies between PEs, we propose the use of a general interconnect that allows for intercommunication between all PEs. A high-level overview of this architecture is shown in Figure 3.1.

3.1.3 Motivation for the selected Hardware Architecture

The direct mapping of task graphs is somewhat simpler in that no extra abstraction layer is introduced, and thus no extra overhead. In addition, the execution time is likely to be close to optimal, since the task graphs are parallelized as much as possible.

However solving ODE systems is to an extent serial in nature, because the next iteration cannot be started before the current is completed. This means that, if separate hardware blocks are to be used for all operations, it is likely that only a few such hardware blocks will be used at any one time. This might not be area efficient, especially for large models.

The PE approach has the advantage that multiple tasks can be assigned to a single PE, if the instruction memory is increased accordingly. This helps with scalability, and reduces area usage. In turn, this can allow very large models to be executed.

In addition the PEs might later be extended to perform other functions, such as non-linear iterative solving, which are not suitable in the direct mapping of task graphs approach.

It was decided that the architecture using connected PEs was to be used in favor of the direct mapping of task graphs.

3.2 System Overview

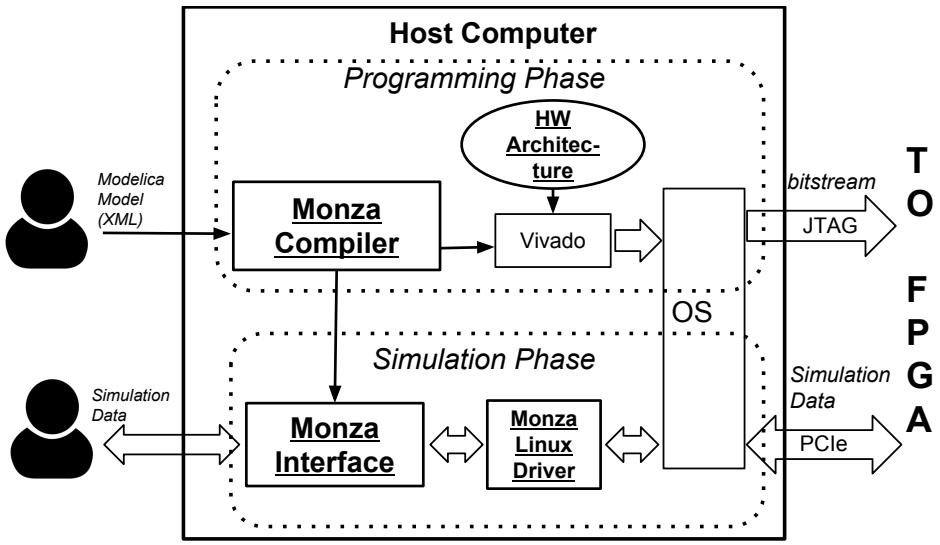


Figure 3.2: *The components developed in thesis, with underlined text, and how they interact with the other components present on the hardware platform. The wide arrows represent interactions through data messages, while the narrow ones represent interactions by reading and writing files. The rectangles in the host computer denote programs running on it, and ellipses denotes files used by programs.*

The name of the developed system is *Monza*, which stands for Modelling ODEs Numerically on the Zynq Architecture. It consists of four components:

1. a hardware architecture, in the form of VHDL files;
2. a compiler written in Python;
3. a user interface, also written in Python; and
4. a Linux driver written in C.

These components can be seen in Figure 3.2.

The interactions of the components and which phase they are used in can be seen in Figure 3.2. These are the same as those shown in Figure 1.2 of the requirements with a few modifications.

In the programming phase, the *Monza Compiler*, in addition to producing hardware configuration files, also produces a simulation parameter file. The hardware configuration files are used to set parameters and the contents of memories present in the hardware architecture files. A more detailed description of this can be found in Section 3.4. The simulation parameter file is used by the *Monza Interface*, and contains the data memory adddresses of the model variables on the FPGA.

In the simulation phase, the simulation interface has been split into the Monza Interface and the *Monza Linux Driver*. The former interprets user commands and executes them by interacting with the FPGA via the latter. The Linux driver, which is a character device driver, enables the communication by implementing POSIX `open`, `close`, `seek`, `read` and `write` system calls. Simulation data is sent between the host computer and the FPGA over PCIe, using a separate PCIe packet for every 32 bit word that is transferred.

3.3 Model Programming Example

Listing 3.1: *Modelica code of the RLC model.*

```
model RLC
  constant Real f_t = 1;
  constant Real R = 1000;
  constant Real L = 0.01;
  constant Real C = 1.0E-7;

  Real u0, uL, uC, uR;
  Real iL, iC, iR, i0;
equation
  u0 = f_t;
  uL = L * der(iL);
  iC = C * der(uC);
  uR = R * iR;

  u0 = uL + uC;
  u0 = uL + uR;
  iL = iR + iC;
  i0 = iL;
end RLC;
```

Listing 3.2: *Snippet of the XML file showing single equation of the RLC model afer transformation by OMC.*

```
<equ:Equation>
  <exp:Sub>
    <exp:Identifier>
      <exp:QualifiedNamePart name=
          "iC"/>
    </exp:Identifier>
    <exp:Sub>
      <exp:Identifier>
        <exp:QualifiedNamePart
            name="iL"/>
      </exp:Identifier>
      <exp:Identifier>
        <exp:QualifiedNamePart
            name="iR"/>
      </exp:Identifier>
    </exp:Sub>
  </exp:Sub>
</equ:Equation>
```

The programming phase involves several steps and might, at first glance, be difficult to comprehend. To combat this, this section presents a short explanation of all steps when programming a specific model for two PEs. This uses the same RLC low-pass filter as in section 2.3.5 from the theory chapter.

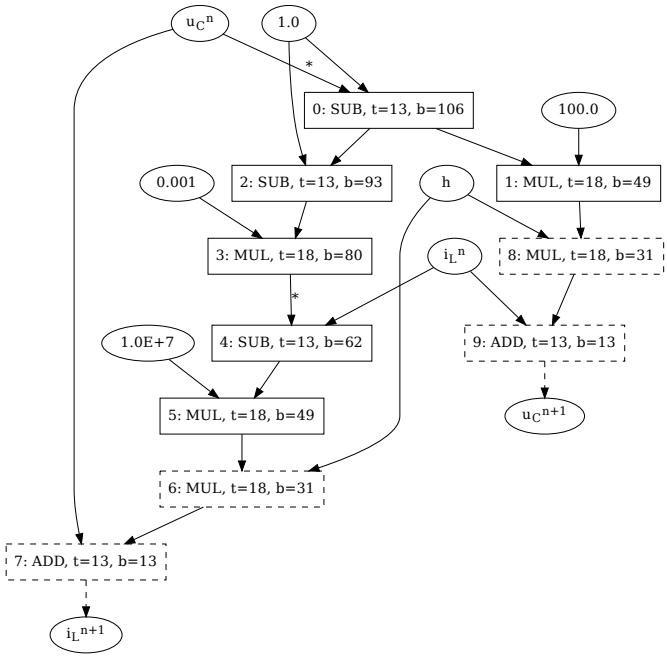


Figure 3.3: *The DAG with operations of the RLC model produced by the compiler. Ellipses represent constants and variables, while rectangles represent operations. The dashed rectangles are used to perform the Euler step. The numbers next to the operations are used to used for identification. The nodes also include t, the execution time, and b, the b-level, explained in 2.9.*

The input consists of modelica code, in this case the modelica file seen in Listing 3.1. The first step transforms the modelica code using OMC which results in an XML-file. This file contains a causalized version of the model and information, such as inital values, about the model variables. This step is analogouos to that performed in section 2.3.5.2.

The second step is to run the Monza compiler on the XML file from OMC. The compiler parses the XML file, and creates a task graph of all the arithmetic operations and their dependencies that are found in the equations. This task graph is shown in figure 3.3. The task graph is augmented with the operations of the Euler step that is used to update the state variables.

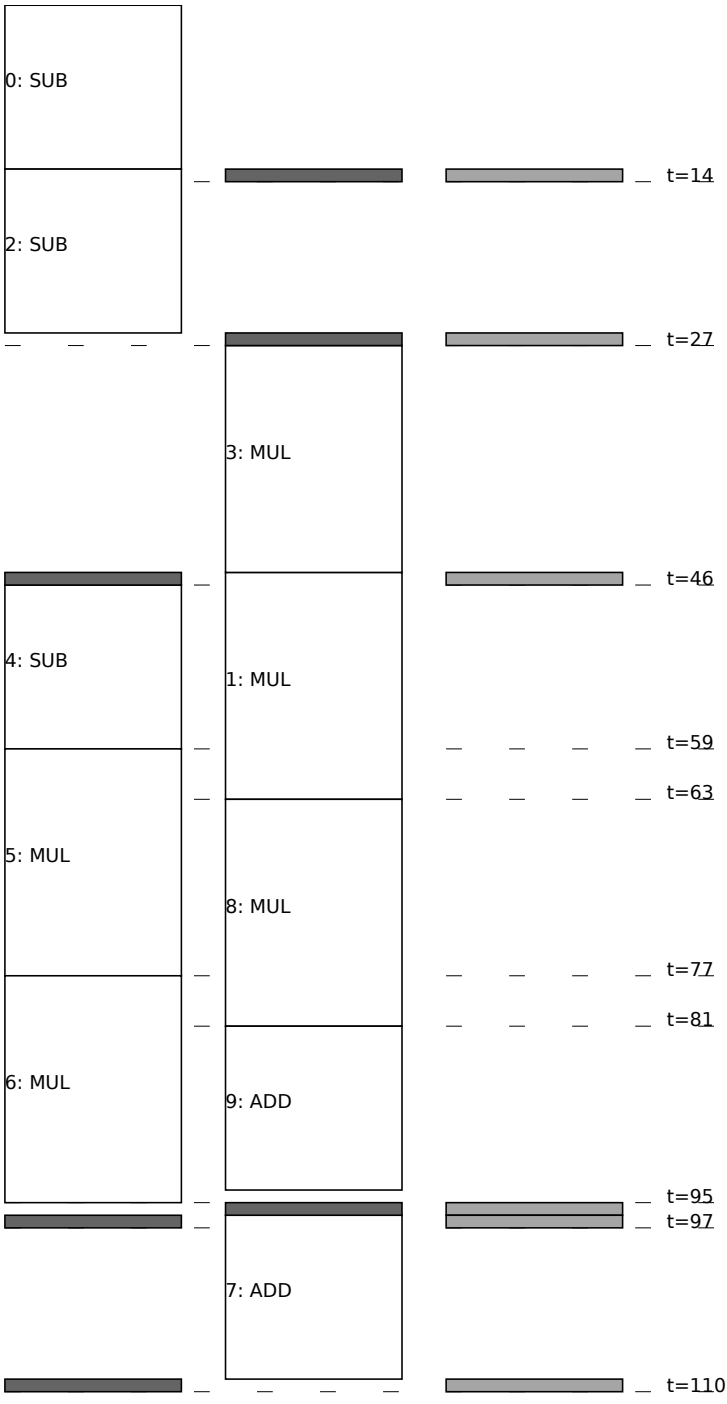


Figure 3.4: *Schedule for the RLC model using two PEs.*

PE0	PE1	Interconnect
SUB 0, 1, 12	WAIT 13	@12, PE0.0, , ,
SUB 0, 12, 11	STORE BS.0, 10	@13, , , PE0.0, BS.0
WAIT 19	WAIT 12	@25, PE0.0, , ,
STORE BS.0, 10	STORE BS.0, 9	@26, , , PE0.0, BS.0
SUB 2, 10, 9	MUL 0, 9, 8	@44, , PE1.0, ,
MUL 3, 9, 8	MUL 1, 10, 7	@45, , , PE1.0, BS.0
MUL 8, 4, 7	MUL 7, 2, 6	@93, , PE1.0, ,
WAIT 1	ADD 6, 3, 3	@94, PE0.0, , ,
STORE BS.0, 2	WAIT 1	@95, , , PE0.0, BS.0
WAIT 12	STORE BS.0, 5	@96, , , PE1.0, BS.0
STORE BS.0, 1	ADD 5, 4, 4	@108, , PE1.0, ,
	WAIT 1	@109, , , PE1.0, BS.0

Table 3.1: *Assembly Code for the RLC model*

Next the compiler uses a scheduler to order the operations of the task graph temporally, and spacially on the different PEs. Figure 3.4 shows how the compiler has scheduled the operations of the RLC model. Note that extra operations, called *store operations*, have been inserted for every bus transfer. This is a consequence of the selected hardware architecture.

Finally, the compiler translates the schedule into the instruction code needed by the PEs and the interconnect. The assembly code for this shown in Figure 3.1, but the compiler also outputs a binary version of the code.

The compiler also outputs the contents of the data memories for each PE, and the *sys_pkg* file 3.3. The latter file specifies the location of the files used to populate the data and instruction memories, the the number of cycles for each iteration, and the number of *write registers* used in the interconnect. The usage of the sys_pkg file is further explained in section 3.4.

Listing 3.3: *The syspkg file for the RLC model. The underlined expressions are generated by the compiler*

```
library work;
use work.pe_pkg.all;

package sys_pkg is
  subtype file_path is string(1 to 57);
  type pe_struct_t is array (0 to 1) of file_path;
  type pe_array_t is array(natural range <>) of
     pe_struct_t;

  constant pe_array : pe_array_t :=(
       (("{BUILD_PATH}/pe_dmem_0.bin",
       "{BUILD_PATH}/pe_imem_0.bin") ,
       ("{BUILD_PATH}/pe_dmem_1.bin",
       "{BUILD_PATH}/pe_imem_1.bin"));
  constant intcon_struct : interconnect_structure_t :=
     (1, 1);
  constant intcon_txt : string :=
     "{BUILD_PATH}/interconnect.bin";

  constant INTERRUPT_CYCLES_INIT : positive := 110*1-1;
  constant PE_ID_HIGH : integer := 17;
  constant PE_ID_LOW : integer := 9;
  constant DMEM_ADDR_HIGH : integer := 8;
  constant DMEM_ADDR_LOW : integer := 0;
end package;
}
```

3.4 Hardware Architecture

Figure 3.5 shows a top-level view of the hardware architecutre that the tesis resulted in. Conforming to the initial design, it contains three types of modules: the PEs, the interconnect and the I/O module.

3.4.1 Top Design

The PEs, interconnect and I/O module are instantiated in a top level design which is imported into Vivado as an IP block and integrated with, for example, low level PCIe and debug cores.

The top level design also instantiates the correct amount of PEs and sets the generic parameters of PEs and interconnect. The instantiation of PEs is done via the VHDL generate construct. The generic parameters are given by a VHDL-file

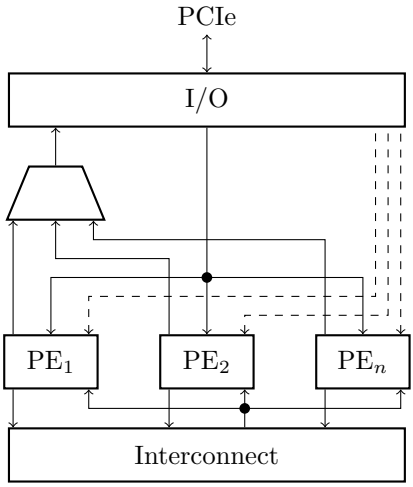


Figure 3.5: *Block diagram of the top design. The dashed lines are control signals from the I/O module to each individual PE.*

containing some constants describing the system. This file is generated by the Monza compiler.

3.4.2 Processing Element

Each processing element is a very simple processor with separate data- and instruction memories. Figure 3.6 shows a block diagram of a PE.

The data memory is 32 bits wide and is implemented as a two port memory, where one port can be overridden by the I/O unit when reading and writing initial values and results. The depth of the data memory is parameterizable and set by the Monza compiler.

The instruction memory is variable width and implemented as read only memory. The instructions are microcode and directly controls all logic in the process element. Figure 3.7 shows an example of a row from the instruction memory. Since the data memory is variable depth, the instruction memory must be variable width to accommodate different address sizes. The instruction set consists of six instructions

1. ADD
2. SUB
3. MUL
4. DIV
5. STORE
6. WAIT

Since the implemented integration method, Euler forward, is explicit, there is no need for the PE to have branching capabilities. The program counter is monotonically increasing during each integration step and has the same value for all PEs. The instruction memory thus has the same length in all PEs even though they might execute a different number of instructions. If that is the case the last instruction is a WAIT, which ensures synchronization.

Each PE takes in the names of two text files with the content of the data memory and instruction memory as generic parameters. This in turn instantiates and initializes the data memory and instruction memory to their minimum depth and width.

The block named FPU seen in Figure 3.6 is an open source single precision IEEE-754 floating point unit which was chosen due to

1. lack of time for developing a specific FPU for this project; and
2. ease of integration compared to Xilinx FPU IPs.

The output from the FPU is constantly fed to the interconnect module where it can be stored in a write register. When a PE requires a value from another PE, the interconnect presents the value on the bus and the reading PE fetches it with a STORE instruction.

3.4.3 Interconnect Module

The interconnect module connects all the PEs' outputs via registers connected to a bus going back into the PEs' inputs. All the logic in the interconnect is controlled directly by microcode in the transfer memory. The number of registers for each PE is variable and set by the Monza compiler. Since the number of registers are variable (and the number of PEs), the number of inputs on the muxes must be variable as well. The interconnect therefore takes two generic parameters upon instantiation, the *interconnect structure* and a text file describing the transfer memory.

The interconnect structure is an array where each element represents the number of registers to generate for the PE at that index. For example the structure $(2, 2, 1)$ would generate 2 registers for PE 1 and 2, and 1 register for PE 3. This would in turn generate 2-input muxes for PE 1 and 2, and no mux at all for PE 3. The final mux would have 3 inputs, since the length of the array is 3. The generation of registers and muxes is done in VHDL via the *generate* construct.

Figure 3.8 shows a block diagram of the interconnect and Figure 3.9 shows an example row from the transfer memory.

3.4.4 Input Output Module

The I/O module is what enables the Monza system to communicate with the host computer. The communication is done on the PCIe bus, over a x4 Gen 1 link. To the host computer, the Monza system looks like a normal peripheral device. Like a graphics- or sound card, the Monza system is presented to the host as a

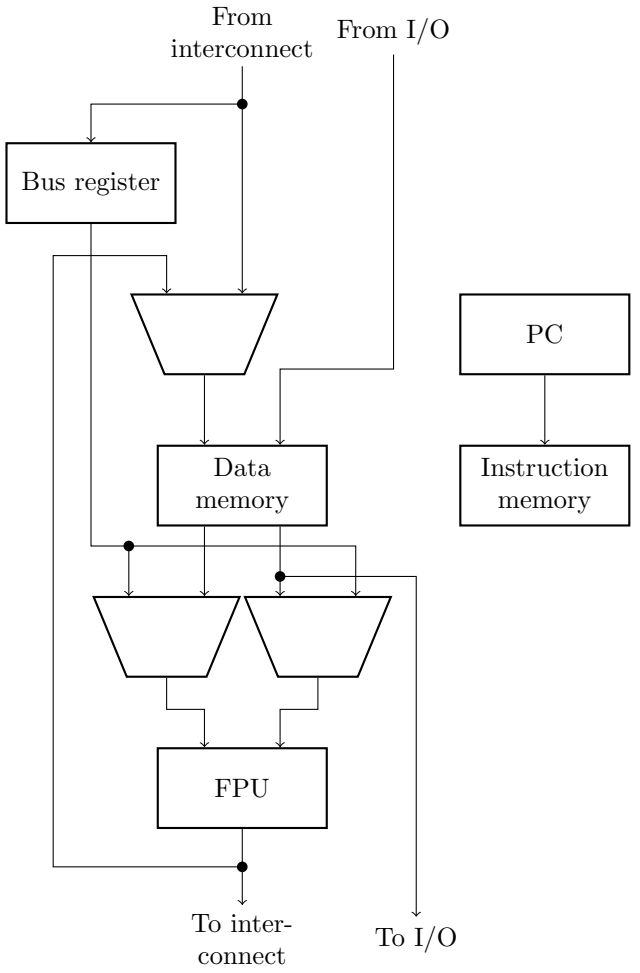


Figure 3.6: *Block diagram of a processing element. Control signals from instruction memory and I/O module have been left out for brevity.*

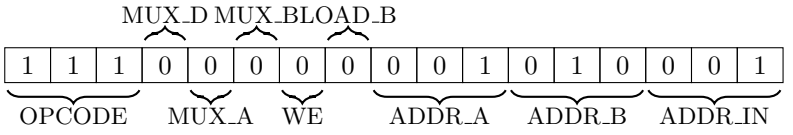


Figure 3.7: *Microcode row from start of a division instruction*

$$DMEM(1) = \frac{DMEM(1)}{DMEM(2)}$$

This PE has a data memory length ≤ 8 since the address fields are 3 bits wide.

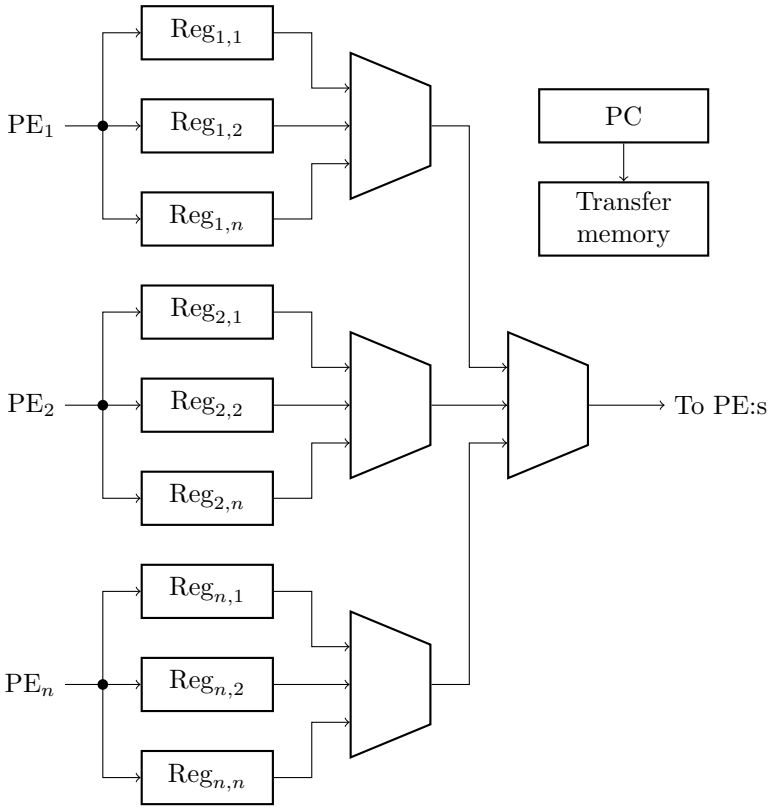


Figure 3.8: *Block diagram of the interconnect. Control signals from transfer memory have been left out for brevity.*

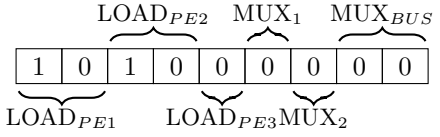


Figure 3.9: *Example row from transfer memory. This interconnect has 2 registers for PE 1 and 2 and 1 register for PE 3.*

region of memory which it can read and write to. Monza also sends interrupts to the host when it is done calculating a configurable number of integration steps. During the simulations in Section 4.4.2 an interrupt is sent after every integration step. For higher performance this can be set to a larger value. However, this will cause a coarse-grained result, where each datum is separated by more than one integration step.

The module consists of a state machine which takes packets form the PCIe bus as input and generates writes or reads to the PEs. In order to simplify the implementation, the I/O module only supports PCIe packets with a payload of 32 bits (one double word) and memory accesses to addresses of multiples of 4. These constraints ensure that the packets are always the same length (4 double words) which cuts down the necessary states and buffers significantly. The address decoding is handled by the driver which makes the memory accesses fairly transparent to the application program.

When a read or write request arrives at the I/O module the lower 11 bits (where the lowest 2 are always 0) of the address are interpreted as the address to the data memory in a PE. Bits 12 to 20 are interpreted as the number of the PE whose memory is to be accessed. If the PE number is all ones, special registers can be accessed, which currently include

1. START - writing to this register makes the Monza system start a calculation, it will run until the configured number of integration iterations is reached

2. N_ITER - writing to this register sets the desired number of iterations

The I/O module works on the *transaction* layer of the PCIe protocol stack. The lower layers, *physical* and *link*, is handled by an IP from Xilinx.

3.5 Software Description

The software toolchain consists of three main parts, the compiler, user interface and driver.

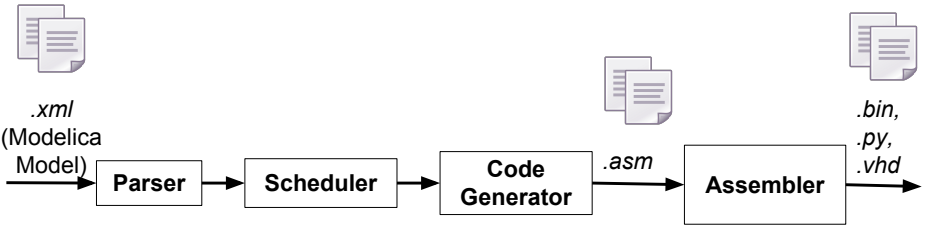


Figure 3.10: *The steps of the Monza compiler.*

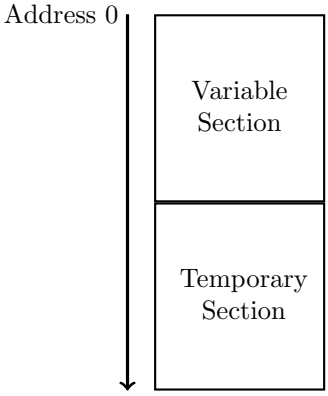


Figure 3.11: *Layout of a data memory.*

3.5.1 Monza Compiler

The steps taken by the Monza Compiler are similar to those found in a traditional compiler [25, p. 5]. The main difference being that the parsing is simplified, thanks to the use of OMC, and that the majority of the work happens in a scheduler. The aforementioned steps are shown in Figure 3.10, and can be described as follows.

First, the parser takes a an XML-file of the flattened Modelica model where the equations have been fully sorted. The compiler parses the equations and builds a DAG for the model with variables and operations as nodes, and the nodes being connected according to how they are used in the equations.

Before the DAG is handed over to the next step, the operations for the Euler step are added. This is done for all state variables in the system. The Euler step consists of two operations: an addition and a multiplication, with the latter being performed after the former. The multiplication operation of the Euler step is referred to in the following sections as an *Euler task*. The result of this operation is used to update the value of the corresponding state variable.

The second step is done by the scheduler, which uses the DAG of the previous step to find a schedule suitable for the given amount of PEs. The scheduling is explained in more detail in the following section.

During the scheduling process, the compiler also allocates memory addresses to the model variables. Currently, variables are duplicated; they are stored in each PE where they are used. Addresses for temporary variables for storing intermediate results of equations are also allocated. Figure 3.11 shows the layout of a data memory: The *variable section*, starting from address 0 contain model variables, the step length, and the constants zero and one. The rest of the addresses form the *temporary section* which contains all temporary variables.

Once a schedule has been found, assembly code for all PEs and the interconnect module is generated. This step produces assembly files which can be used to debug the system.

The last step is performed by the assembler, which assembles the assembly files into binary files. Thes will ultimately be synthesized into the instruction memories of the PEs and into the transaction memory.

3.5.2 Scheduling

In the following section the term *task* refers to an operation that is to be or is part of a schedule.

The compiler uses a the list scheduling approach as mentioned in Section 2.9 with one modification at step 2: Instead of selecting a PE that allows the earliest start time, a PE whose last task finished earliest is selected, and the task is scheduled as early as possible on that PE. Depending on the implementation this method may lead to faster compile times for a large number of PEs. However it is likely to generate longer schedules because the search-space becomes smaller.

The behaviour of the scheduler follows the flowchart shown in Figure 3.12. The steps therein can be described as follows:

1. The b-level, as defined in Section 2.9 is calculated for each node in the DAG.
2. The entry tasks, as defined in Section 2.9, of the DAG are added to the ready list.
3. If there are still items in the ready list, select the task, `T`, with a lowest b-level, breaking ties randomly.
4. If the ready list is empty, the scheduling is complete.
5. `T` is assigned to a PE, `P` if it has not already been assigned to one.
6. If `T` references a state variable, a store task `S` is created with the output from the corresponding Euler task as input. Also, `S` is assigned to PE `P`. The reason for creating this new task is to ensure that the state variables are updated for all the PEs in which they are used.
7. Stores for dependent tasks of `T` are created and scheduled immediately. The purpose of these tasks is to transfer the result of tasks that are computed on PEs other than that of `T`.
8. `T` is scheduled as early as possible, but after all dependencies are ready, on PE `P`.
9. Finally, tasks that depend on `T` that become have ready as a result of scheduling `T` are added to the ready list.

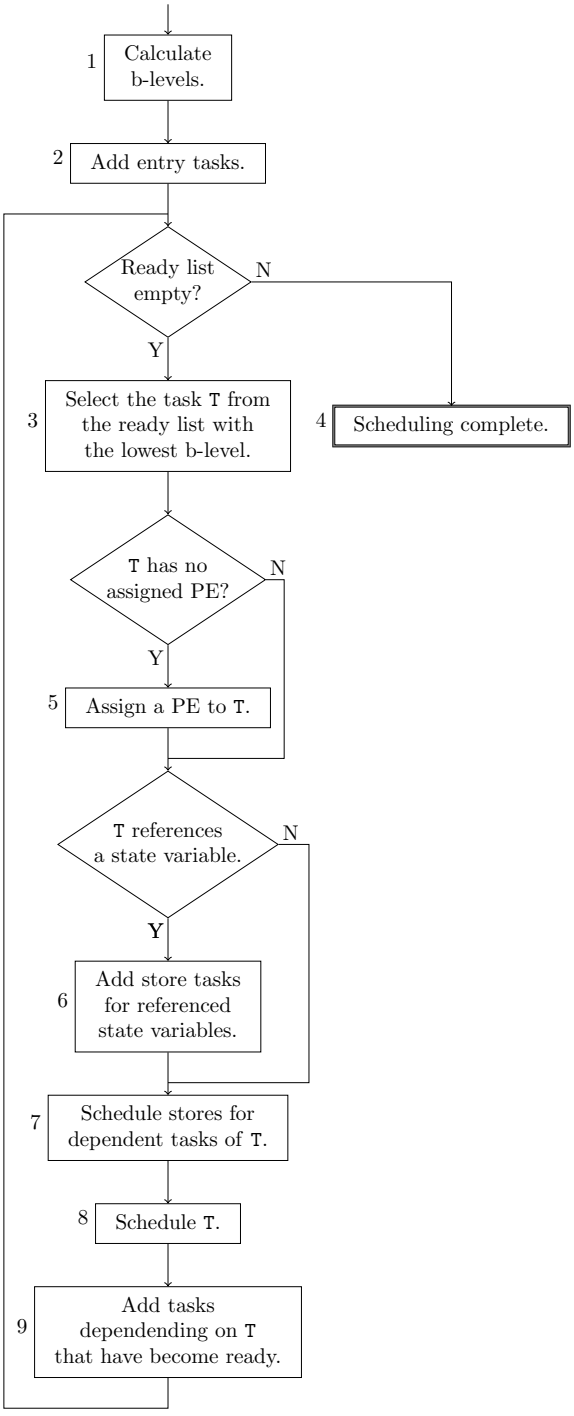


Figure 3.12: *Flowchart of the scheduling algorithm used in the compiler.*

3.5.3 Monza Interface

Descriptive Name	Variable Type ID	Description
Unknown	-1	A variable whose type could not be identified
Parameter	0	A Modelica parameter
Constant	1	A constant
(Unused)	2	
(Unused)	3	
(Unused)	4	
State	5	A state variable
Derivative	6	A derivative variable
Monza Generated	7	A variable that has been inserted by the Monza compiler

Table 3.2: *Variable Type IDs with descriptions.*

the Monza Interface program can be used for manual reading and writing, or for running simulations. When used in the former mode, the interface allows retrieval and modification of variables. The corresponding commands require a PE number and an address in the data memory of that PE. The user must thus keep track of which memories (a variable may be stored on multiple PEs) and on what addresses the variables of interest are stored.

Fortunately, variable addresses can be retrieved from a simulation parameter file called `variables.py`, that is outputted by Monza compiler as mentioned in section 3.2. This file contains an array of the variable section for each data memory. Each variable section is represented as an array where the index corresponds to the address in the data memory. The elements are tuples of variable name, as read from the OMC XML-file, and an integer denoting the variable type. The different variable types are listed in Table 3.2.

The file `variables.py` is always used when running a simulation. When doing so, the Monza Interface program uses another data structure present in the file, containing actions that should be performed at specific clock cycles. The data structure is a dictionary with clock cycles as keys and actions as values. The actions (there can be multiple actions per clock cycles) for a specific clock cycle are stored in a dictionary with action types as keys. The two types of actions are:

1. `write`, whose corresponding value is an array of variables to be written to, with each element being a tuple containing the PE to write to, the address on that PE and the value to be written.

2. `read`, with an array variables to read from, each element being a tuple of PE and PE address where the variable is located.

The Monza Interface, in addition to performing the actions in `variables.py` also reads the values of all state variables and stores them in a file called `monza_res.dat`

when the simulation has finished.

3.5.4 Monza Linux Driver

The Monza driver is a Linux driver written in C. The function of this driver is to create a character device when a Xilinx ZC706 card is inserted into the host computer, and it supports the operations:

- `open`,
- `release`,
- `llseek`,
- `read`, and
- `write`

The `open` function associates a Linux `file` structure with the character device for the card, thereby allocating a file address for it.

The `read` and `write` operations respectively send and request a 32-bit word to and from the card, using the file position as the address field in the PCIe packet.

An exception is made when writing the start command. In this case, a ready flag is set indicating that the driver should wait for an interrupt. Upon performing a read, the driver is put to sleep if the ready flag is still set. The same flag is unset in an interrupt handler associated with the card, which wakes up the driver in case it has been put to sleep.

4

Experimental Results

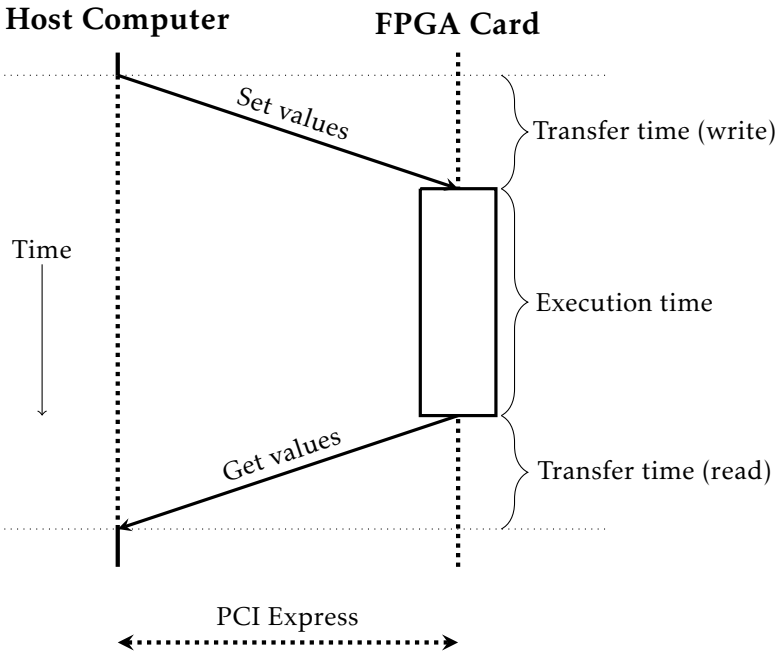


Figure 4.1: *Experiment setup for the FPGA system.*

Since this thesis primarily deals with acceleration of existing numerical methods, *execution time* was used as the primary performance metric. The execution time was compared to existing solutions, which in this case was model simulation using OpenModelica with a single-threaded process.

Apart from the execution time, the correctness of the results was evaluated. The results from the new system were compared to a known good reference simulation on a CPU. This is important since the execution time is irrelevant if the simulation results deviate too much from the reference.

To help in the search for bottlenecks, two additional aspects were measured: the fraction of time spent doing store operations, and the fraction of time spent idling. The store time is an overhead that is imposed by the system. On the other hand, the idle time is affected by how parallel the model is, how good the scheduler is, and how saturated the bus is.

On the FPGA system, the *transfer time* is also important. Since initial values and results must be transferred from main memory to the FPGA and back, this adds to the total simulation time. The definition of transfer time is illustrated in Figure 4.1 and includes both reading and writing.

Due to the scope of the thesis, the experiments did not include test models with general input functions. However, the models were run with different initial values.

4.1 Experiment Setup

In order to examine how the system performs under different circumstances, two parameters were studied: the *model size* and the *number of PEs*. These two parameters make up the *configuration* of each experiment. The number of PEs only affects the FPGA measurements, since it is not applicable to the CPU. How the model sizes are varied can be read in Section 4.2.1 and 4.2.2.

During all measurements, the integration method used is Euler forward and the simulated time is 1 second with $h = 0.0005$, resulting in 2000 steps.

Details of the hardware platforms and toolchains used in the experiments are summarized in Table 4.1 and 4.2. Pseudocode for the testing procedure can be found in Listing 4.1.

Listing 4.1: *Testing procedure.*

```
for model_size in model_sizes:
  for n_pe in 2,4,6,8,10,12,14,16:
    set_model_size(model_size);
    compile_model();
    simulate_on_cpu();
    run_fpga_model_compiler();
    run_fpga_toolchain();
    simulate_on_fpga();
  end for;
end for;

generate_plots();
```

FPGA Board	Xilinx ZC706
FPGA Model	Z-7045
FPGA LUTs	218600
FPGA Block RAM	19.2 Mb (545 * 36 Kb)
HDL	VHDL
Toolchain	Vivado® Design Suite: Design Edition 2015.1
OMC	
Version	1.12.0-dev.alpha1
Flags	`-s --simCodeTarget=XML`
Host Computer	
Model	Dell Precision T7500
CPU	Intel Xeon X5647 @ 2.93 GHz
Operating System	CentOS 7

Table 4.1: *FPGA system specifications.*

Model	Dell Precision T7500
CPU	Intel Xeon X5647 @ 2.93 GHz
Operating System	CentOS 7
OMC	
Version	1.12.0-dev.alpha1
Flags	`-s --simCodeTarget=C`

Table 4.2: *CPU system specifications.*

4.2 Test Models

In order to measure the performance of the system, test data is needed. In the case of model simulation, test data are models. Since the variation of possible models are as large as the engineering field itself, some limitations must be imposed. The test models should:

- reflect real world systems and conditions,
- be complex enough to exercise all parts of the FPGA system, and
- be scalable.

Two models were selected for this purpose. The code for them can be found in Appendix B.

4.2.1 Weibel Lung

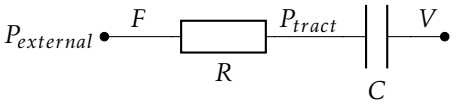


Figure 4.2: *Model of an air tract in the Weibel lung.*

The Weibel lung, also studied in [17], is a model of a human lung which works by treating the lung as a full binary tree RC-network. Each node of the tree, which corresponds to an air tract, is represented by a resistor in series with a capacitor, as can be seen in Figure 4.2. Figure 4.3 shows a three-generation weibel lung.

The equations for each Weibel node are as follows:

$$F = \frac{(P_{external} - P_{tract})}{R}$$

$$P_{tract} = \frac{V}{C}$$

$$\frac{dV}{dt} = F$$

Where $P_{external}$ is the pressure external to the tract, either from an adjacent tract or from the mouth, and P_{tract} is the pressure inside the tract. F is the air flow into the tract. R and C is the resistance and capacitance, respectively, of the tract.

The model size parameter for this model controls the number of generations in the tree. Increasing model size trees gives a more accurate description of the lung, at the expense of increasing the number of ODEs exponentially. The relation between the number of generations, or N, and the number of state variables is shown in Table 4.3.

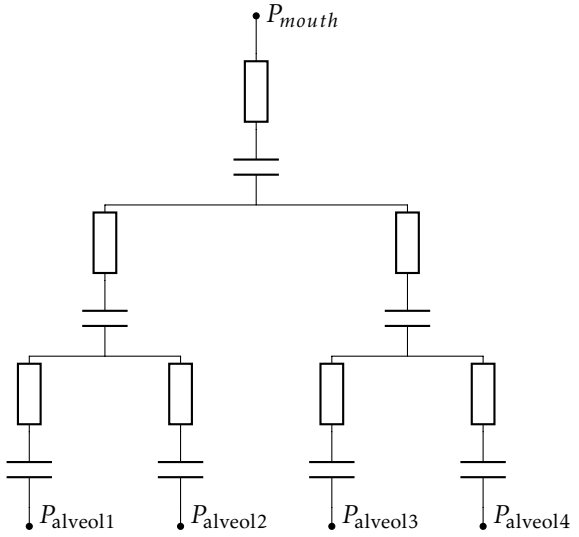


Figure 4.3: *A three-generation Weibel lung. The alveols are the outermost parts of the lung.*

N	#State Variables
2	2
3	6
4	14
5	30
6	62
7	126
8	254
9	510
10	1022
11	2046

Table 4.3: *Relation between N and the number of state variables for the Weibel model.*

4.2.2 Water Tank System

The water tank model is developed specifically for this project since modeling fluid in pipes and containers is interesting from the stakeholder perspective (see Section 1.6).

The model consists of a variable number of tanks connected with pipes, as can be seen in Figure 4.4.

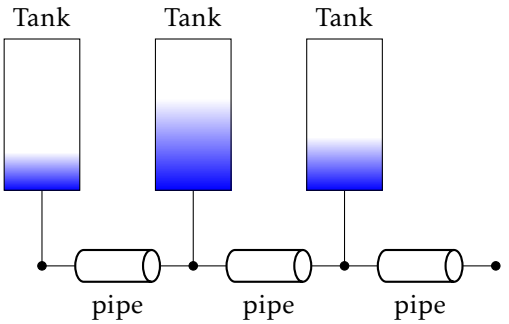


Figure 4.4: *Water tank system model.*

The system is built from separate models for pipes and tanks which are connected by Modelica *connect* constructs. This is similar to how a real model is made in the industry, which motivates the inclusion of this model.

The pipe is modeled by an inertance in series with a resistance, each having two ports where flow can occur. The tank is modeled by a cylindrical container with a single port at the bottom. The equations are therefore divided into three sets:

1. Inertance

$$P_{diff} = P_b - P_a$$

$$P_{diff} = \frac{L\rho}{A_p}\frac{dQ_a}{dt}$$

$$0 = Q_a - Q_b$$

   Where P_{diff} is the pressure difference between the ports A and B. L is the length of the pipe and A_p is the cross-sectional area. Q_a and Q_b is the flow at port A and B. ρ is the fluid density.

2. Resistance

$$0 = \frac{1}{R}(P_b - P_a)$$

$$0 = Q_a - Q_b$$

   Where R is the resistance of the pipe. P_a, P_b, Q_a, Q_b as above.

3. Tank

$$P = \rho g l$$

$$Q = A_t \frac{dl}{dt}$$

Where ρ is the fluid density. g is gravitational acceleration. l is the water level in the tank. A_t is the cross-sectional area of the tank. P and Q are the pressure and flow at the port of the tank.

The model size parameter controls how many tanks and pipes are connected in series. This causes a linear increase in the number of equations.

N	#State Variables
2	3
4	7
8	15
16	31
32	63
64	127

Table 4.4: *Relation between N and the number of state variables for the Tanks model.*

4.3 Clock Frequency

The clock frequency of the Monza system is 125 MHz and generated directly from the Xilinx PCIe IP. The next step up in frequency is 250 MHz which gave timing errors on larger models.

4.4 Correctness

4.4.1 Method

The correctness was measured using the Chebychev distance of the same state variable from the CPU simulation and the FPGA simulation, during all time steps, as in Equation 4.1.

$$\text{Error} = D_{Chebychev}(\mathbf{c}, \mathbf{f}) = \max_i(|c_i - f_i|) \tag{4.1}$$

where $\mathbf{c}$, $\mathbf{f}$ are vectors representing the value of a state variable during all time steps from the CPU and FPGA respectively. This calculation was done for all state variables and then presented in a box plot.

4.4.2 Results

Figure 4.5 shows the calculation error when running the *Tanks* model on the FPGA with different model sizes. The ends of the box plot shows the minimum and maximum value. Figure 4.6 shows the same information for the *Weibel* model. Table 4.5 and Table 4.6 shows the mean and variance.

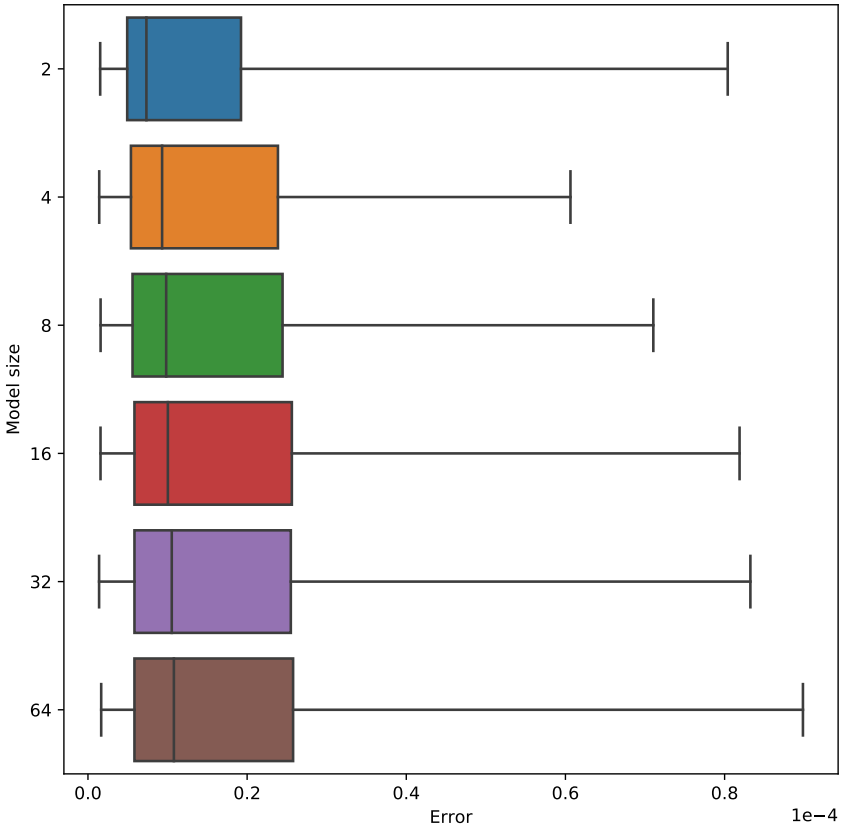


Figure 4.5: *Calculation error for the Tanks model.*

Model size	Sample Mean	Unbiased Variance
2	0.000015	2.608473e-10
4	0.000016	1.666394e-10
8	0.000016	1.597477e-10
16	0.000017	1.700176e-10
32	0.000017	1.634024e-10
64	0.000017	1.614600e-10

Table 4.5: *Mean and variance of calculation errors in Tanks model.*

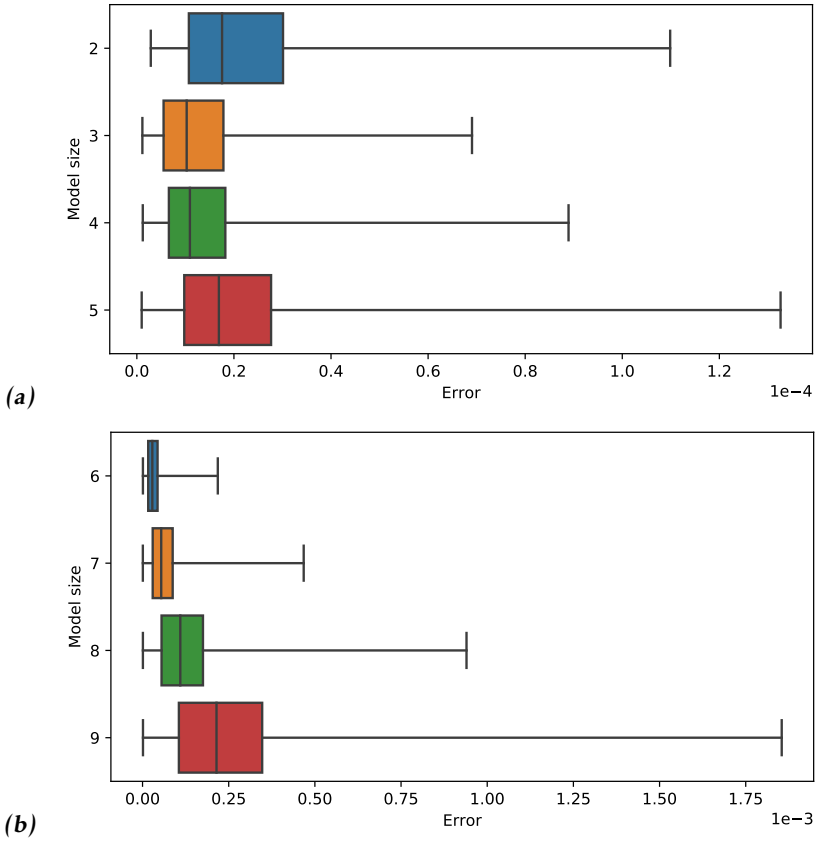


Figure 4.6: *Calculation error for the Weibel model. Notice the different scales for the small and large model sizes respectively.*

Model size	Sample Mean	Unbiased Variance
2	0.000023	3.446314e-10
3	0.000014	1.435466e-10
4	0.000015	1.637458e-10
5	0.000022	2.979328e-10
6	0.000034	6.678337e-10
7	0.000065	2.504160e-09
8	0.000129	1.034542e-08
9	0.000256	4.254999e-08

Table 4.6: *Mean and variance of calculation errors in Weibel model.*

4.4.3 Discussion

For the Tanks model, no clear trend can be seen in the error. Neither increasing or decreasing, it appears to remain fairly constant at around 0.00001 with maximum values of 0.00009 for all configurations. This is quite small, and in fact close to the limit of single precision arithmetic, which is 7 significant decimal places [26, p. 4]. However, the required correctness depends heavily on use case.

The Weibel model, on the other hand, shows a clearly increasing trend in the error values. The error is fairly constant up to and including model size 6. The error roughly doubles for each size after size 7.

The most likely cause of the errors are that the CPU simulations are performed using double precision arithmetic, which has 15 significant decimal places [26, p. 4]. Therefore, the error is likely to increase as the model size and the number of iterations grows.

Another possible source of errors in the FPGA system is that operations can be performed in another order than on the CPU, due to parallel scheduling.

For a more fair comparison, the CPU simulations should have been done in single precision, or a double precision FPU been used instead.

The results for the larger configurations of the Weibel model clearly shows that a double precision FPU is needed for the system to be usable for large models.

4.5 Execution Time

4.5.1 Method

The execution time for the FPGA system is deterministic and can thus be determined by running the compiler and examining the generated instructions. The length of the instruction memory is equal to the number of cycles the FPGA has to execute. Since the clock frequency is a known constant, the number of cycles times the clock period was used as the execution time.

The compiler for the FPGA system uses randomness (see section 3.5.2, step 3.) when scheduling instructions, which results in slightly different schedules and thus instruction memory length for different random seeds. Therefore 10 seeds were tried for each configuration, and the one resulting in lowest number of instructions is presented.

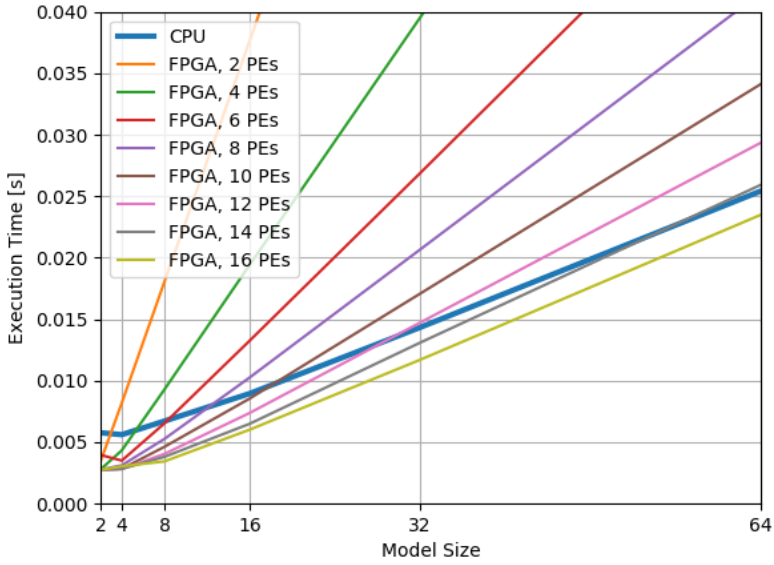
Unlike the FPGA system, the CPU simulation time is the average of at least 8 simulations for every configuration. This was done to account for variances when running programs on a CPU.

For the simulation on the CPU, the execution time was taken from the *SIMULATION TIME* output of when using the `-lv=LOG_STATS` flag with the OMC simulation executable.

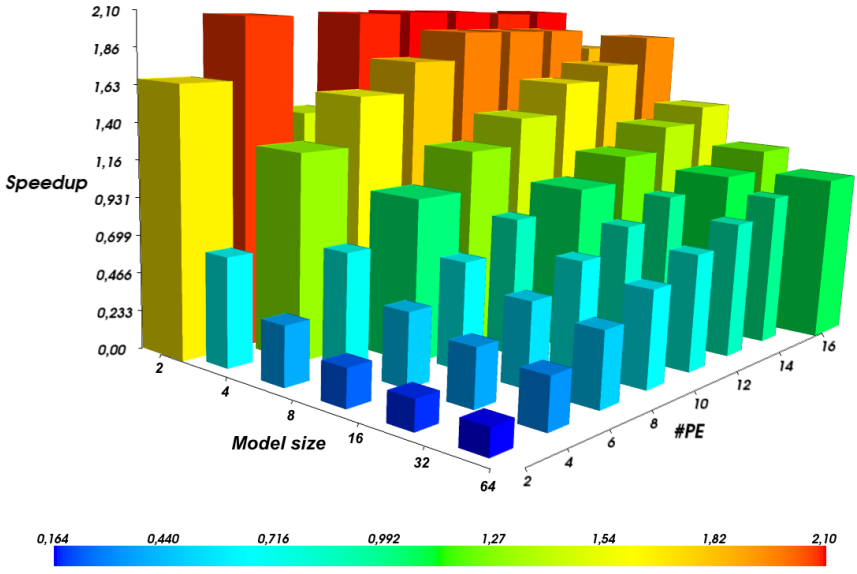
The speed up is measured using the following forumala:

$$\text{Speed up} = \frac{T_{\text{CPU}}}{T_{\text{FPGA}}}$$

4.5.2 Results

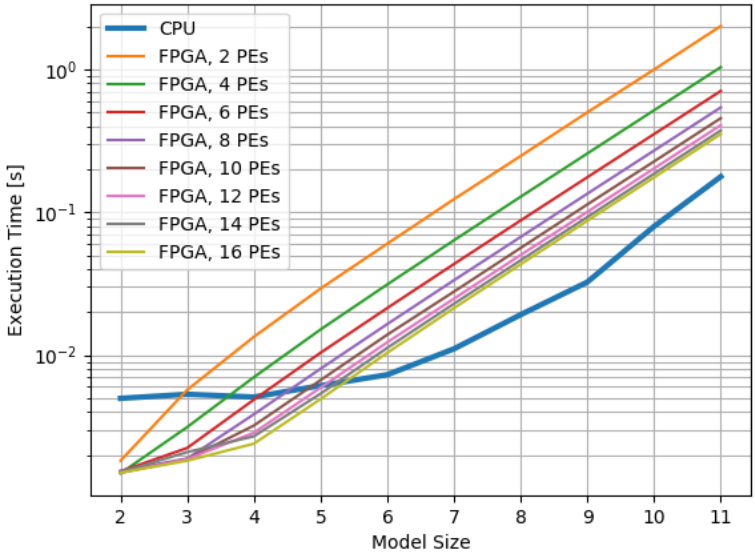


(a) CPU and FPGA absolute execution time.

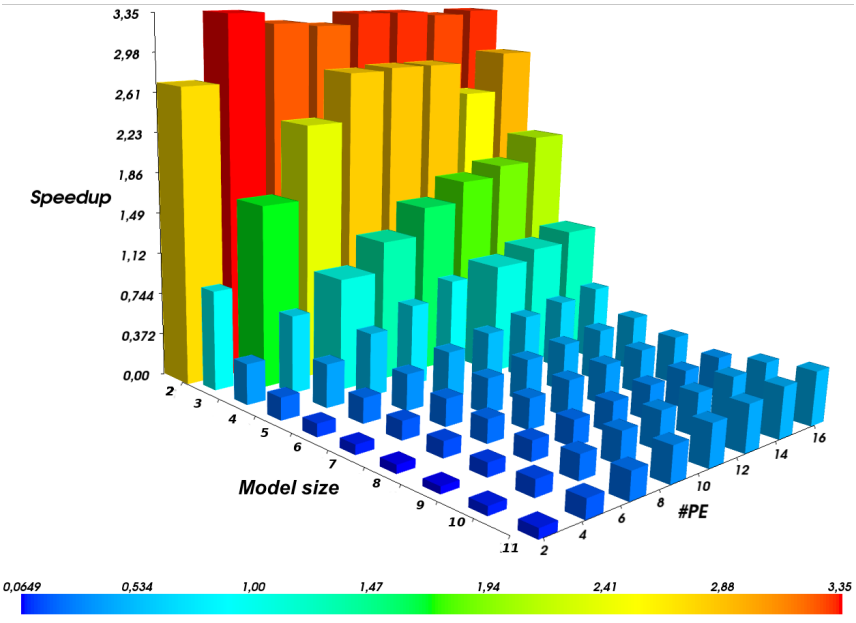


(b) FPGA speed up

Figure 4.7: *Execution time for the Tanks model.*



(a) *CPU and FPGA absolute execution time.*



(b) *FPGA speed up.*

Figure 4.8: *Execution time for the Weibel model.*

Figure 4.7 shows the execution time for the Tanks model. In Figure 4.7a the time taken for simulation on the CPU and the FPGA can be seen, while Figure 4.7b shows the relative speedup (or slowdown) when executing the model on the FPGA. In the latter figure, thin bars indicate a speedup of less than 1 – a slowdown. Figure 4.8 shows the same information for the Weibel model.

4.5.3 Discussion

Figure 4.7a show that the FPGA with 16 PEs is slightly faster than the CPU for all model sizes of the Tanks model. For the Weibel model, Figure 4.8a show that the FPGA with 14 and 16 PEs are faster than the CPU for model sizes up to 5. In general, the greatest speedup is achieved for smaller models.

According to the Figures 4.8b and 4.8b the speedup for the models scale approximately linearly with the the number of PEs for all but the smallest model sizes. The lack of speedup for the smaller models is very likely due to them only being parallelizable to a small degree.

An interesting observation is that the relative speedup decreases as the model size increases. This is caused by the fact that the FPGA execution times, while initially being lower, have a greater slope when increasing the model size, compared to that of the CPU.

The reasons why the FPGA does not scale as well as the CPU, are likely that FPGA has a lower clock frequency, and that the FPU of the FPGA is not as efficient as that of the CPU. The latter is, in fact, further supported when looking at the store and idle times as explained in Section 4.6.

On the other hand, the reason that the CPU starts with a lower execution time for smaller models may be because it requires time to set-up. For example due to initial latencies caused by the OS, and that the caches need warm-up.

Regarding the speed of the CPU, it is in fact twice as fast as the FPGA for the larger models. But, this is when the FPGA system is operating at the relatively low frequency of 125 MHz. The clock frequency should be able to go quite a lot higher with proper placement of the logic. During this project the placement has been done automatically by Vivado, and the FPU contained in every PE has quite long paths in it, resulting in a sub-optimal performance.

4.6 Store- and Idle Time

4.6.1 Method

For the store time, the number of store operations across all PEs were calculated for each configuration. This was divided by the total execution time, derived in the manner explained in the previous section. The idle time was calculated in the same way as the store time, except for the fact that NOPs were counted instead of store operations.

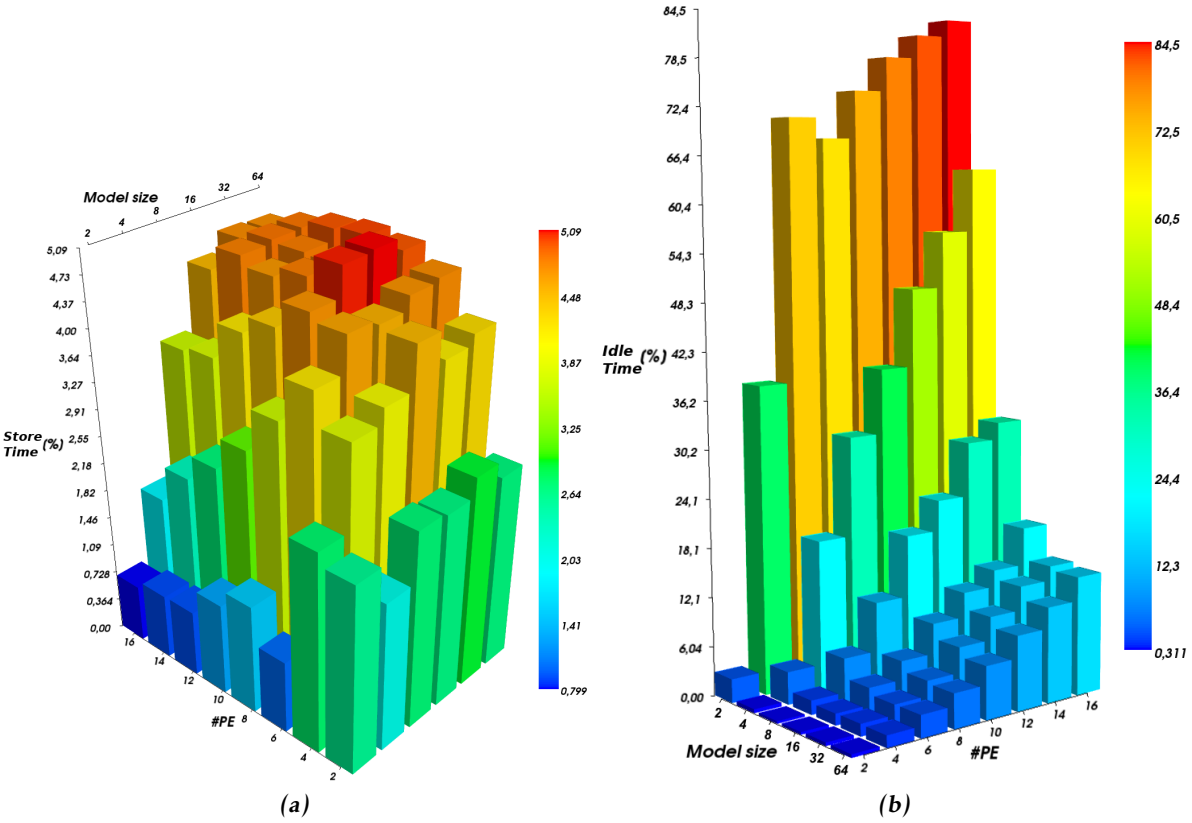


Figure 4.9: *Store and idle time for the Tanks model.*

4.6.2 Results

Figure 4.9a shows the percentage of time spent on store instructions compared to the total execution time for the Tanks model. Figure 4.9b shows the percentage of time spent idling compared to the total execution time. Figure 4.10 shows the same information for the Weibel model.

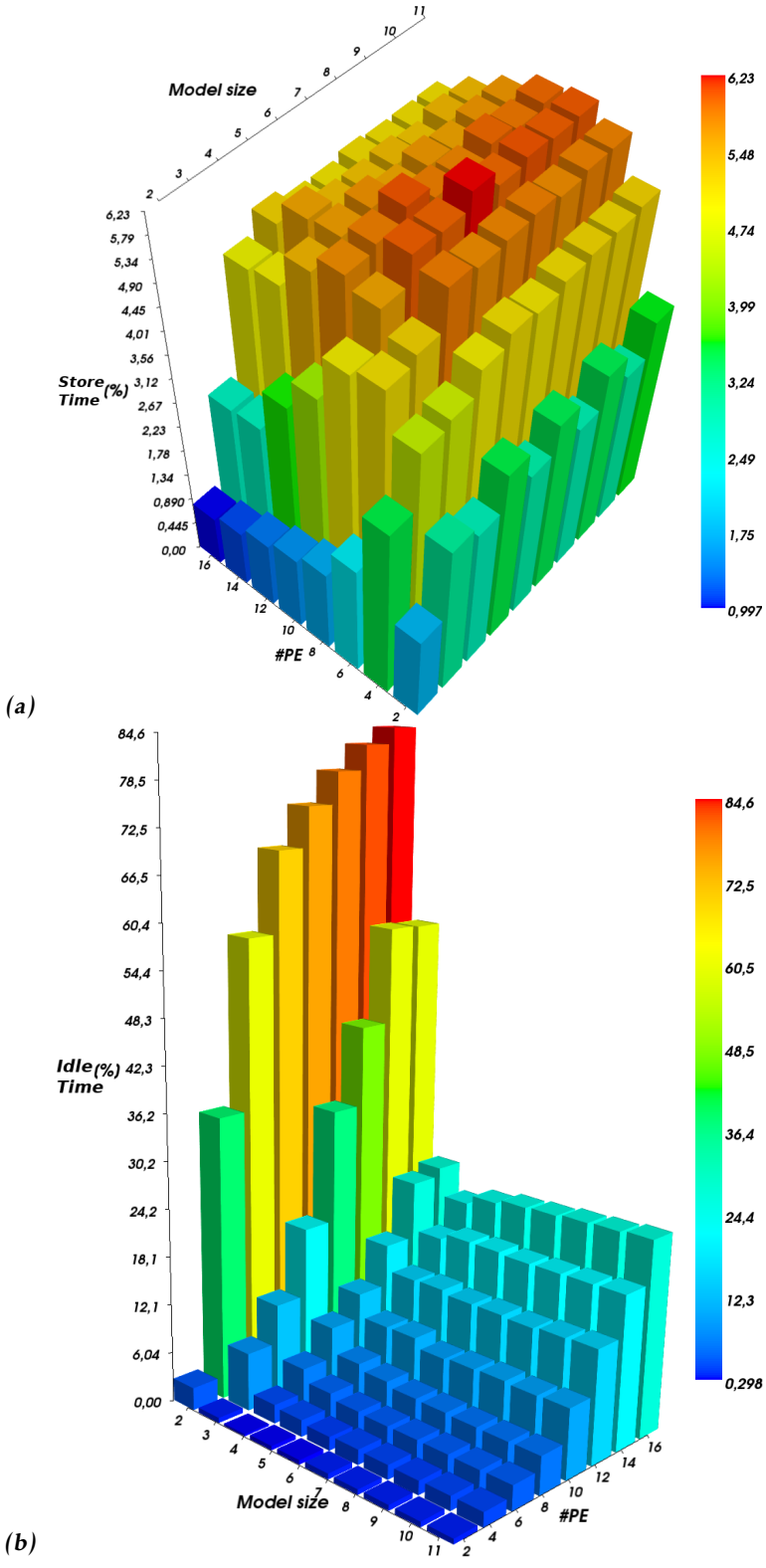


Figure 4.10: *Store and idle time for the Weibel model.*

4.6.3 Discussion

An indication of which future improvements could result in the greatest reduction of execution time, can be gathered from an analysis of the percentage of the time spent doing computations, store operations and idling. Figures 4.9a and 4.10a show that the time spent in store operations is less than seven percent for all configurations. For the idle time, Figures 4.9b and 4.10b show that, except for some outliers that are discussed in the following section, the idle time is less than 30%.

Subtracting the time spent idling and storing from the total, it is evident that the difference, which is computing, accounts for most of the time. Improving the FPU should therefore be of top priority.

Furthermore, Figures 4.9a and 4.10a shows that: for increasing numbers of PEs, the number of stores generally increases up to a point then decreases. At first this can seem unintuitive, since a larger number of PEs should lead to an increase in bus traffic. This may however be explained by increases in idle time: For a majority of the configurations, the percentage of idle time is greater than that of store time.

On the other hand, the percentage of store time is not as much affected when increasing the model size. Apart from a sharp increase for smaller models the proportion of store time remains constant for medium and larger sized models, given the same number of PEs.

When looking at the idle time, there is a sharp increase for the smallest model sizes when increasing the amount of PEs. This may be due to the fact that these models are not very parallel. This is also reflected in the speed-up not scaling well, as explained in the previous section.

As the model size increases, however, idle times appears to only be affected by the number of PEs. For these configurations, increasing the number of PEs correlates with a steady increase in idle times, up to around 24% for the Weibel model, and around 20% for the tanks model.

The reason for the increase in idle time may that the number of bus transfers increases as the number of PEs increase. In turn, these transfers may lead to the bus being saturated, thus causing a bottleneck.

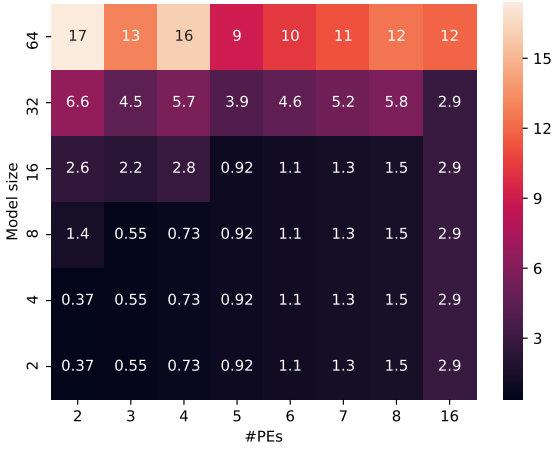
4.7 Utilization

4.7.1 Method

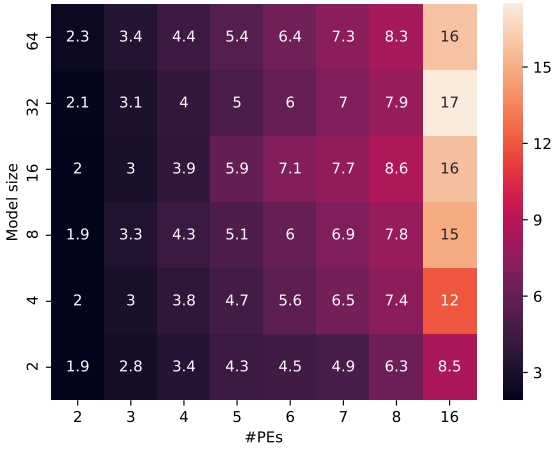
The utilization of the FPGA resources was collected directly from the Vivado log output, where it is given as a percentage.

Since the fraction of data, instruction and transaction memory cannot be directly read from the logs, they were calculated as the percentage of the theoretical maximum block RAM storage on this device, given in Table 4.1. The theoretical memory usage is calculated by counting the number of bits in the transaction, data and instruction memory outputted by the Monza compiler. The data- and instruction memory size is calculated as the total for all PEs.

4.7.2 Result



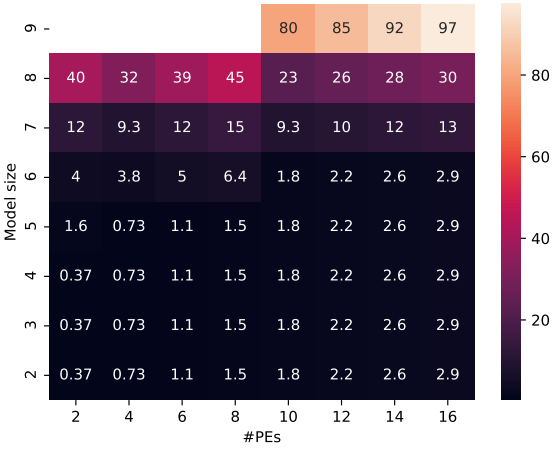
(a) *Block RAM.*



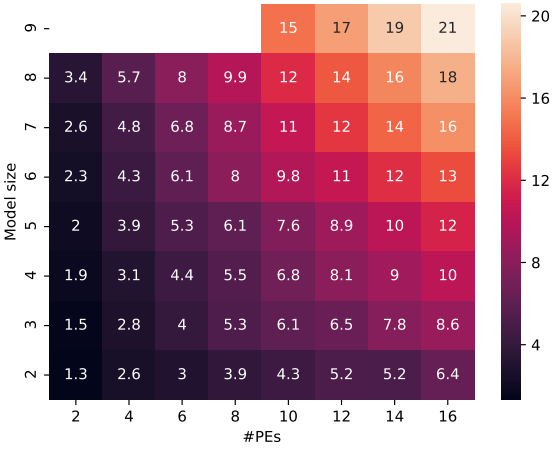
(b) *Slice LUTs.*

Figure 4.11: *Utilization for Tanks model. Values are given as percentage of maximum utilization as reported by Vivado.*

The utilization of the resources on the FPGA depends on how big the model is and the chosen amount of processing elements. More processing elements means more logic, while the model size affects the length of the data memories and thus increases block RAM usage. Figure 4.11 shows the resource utilization for the Tanks test model. Figure 4.12 shows the resource utilization for the Weibel test model. Figure 4.13 shows the theoretical memory requirements for the Weibel model.

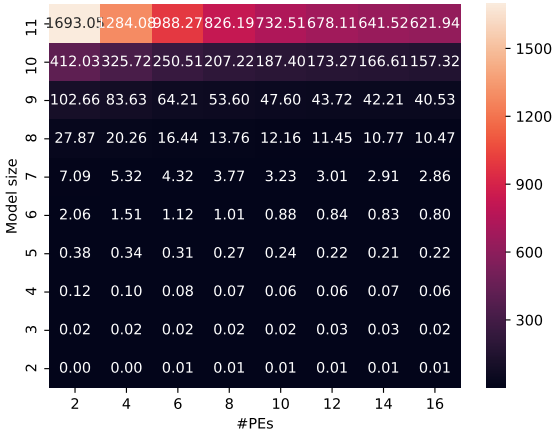


(a) Block RAM.

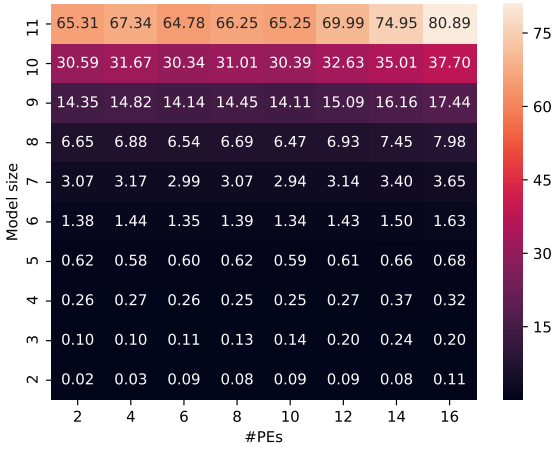


(b) Slice LUTs.

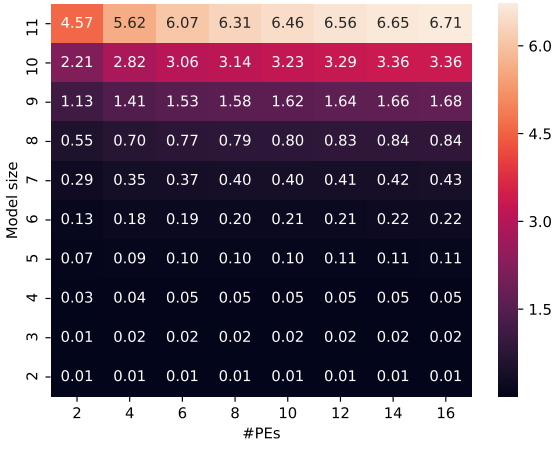
Figure 4.12: *Utilization for Weibel model. Values are given as percentage of maximum utilization as reported by Vivado.*



(a) Transaction memory.



(b) Instruction memory.



(c) Data memory.

Figure 4.13: *Memory requirements for the Weibel model. Values are given as percent of theoretical maximum block RAM storage (19.2 Mb). Keep in mind that the largest models are not synthesizable. This explains why their total is over 100%.*

4.7.3 Discussion

From Figures 4.11 and 4.11 it can be seen that the limiting factor for how large models that can be synthesized is block RAM usage. For the Weibel model, Figure 4.12b indicate that logic resources are not a problem, while Figure 4.12a show that block RAM utilization is almost 100% for the larger configurations.

In fact, the largest model sizes, 10 and 11, could not be synthesized at all. For model size 9 only half of the configurations could be synthesized. The cause of this was found in Vivado logs, which stated too high block RAM usage. Figure 4.13 verifies this, since the larger model sizes uses a significant amount of memory, specifically transaction memory as seen in Figure 4.13a. It seems like the transaction memory usage is inversely proportional to the number of PEs in the system. This would explain why the configurations of model size 9 with #PE < 10 failed to synthesize.

A reason for the transaction memories being very large, is an overabundance of NOPs. One way to solve this would be to store transactions in some kind of associative memory. This way, no NOPs would have to be stored at all. Another way to reduce transaction memory would be to investigate different instruction sets that reduce the width. The transaction memory is, in fact, by far the widest of the memories in the current system.

4.8 Transfer Time

4.8.1 Method

Because the transfer time is highly dependent on the number of variables that user is interested in, transfer time was measured separately from execution time. This was done by measuring the time taken to perform a number of I/O operations. The test program used in the procedure is found in Listing A.1 in Appendix A.

4.8.2 Results

The amount of I/O needed depends heavily on how big the model is as. This can be seen in Figure 4.15, which shows the number of operations needed to read from and write to all variables of the Weibel model for different configurations. Figure 4.14 shows the time required for different number consecutive reads and writes.

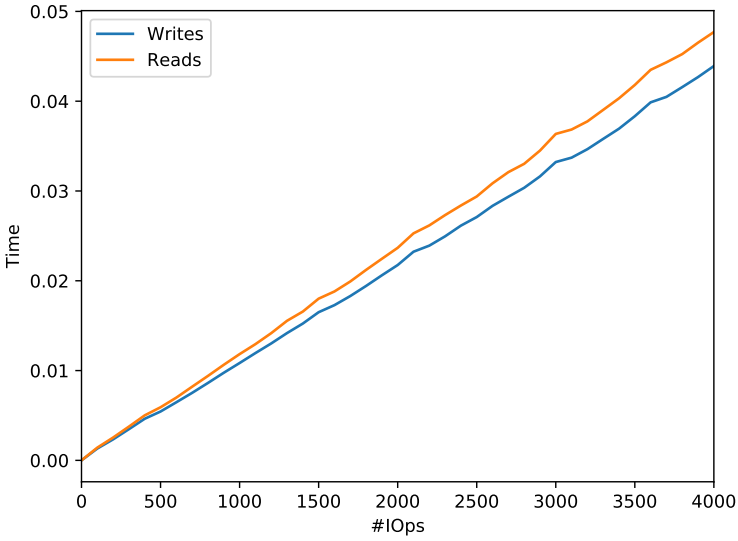


Figure 4.14: *I/O performance.*

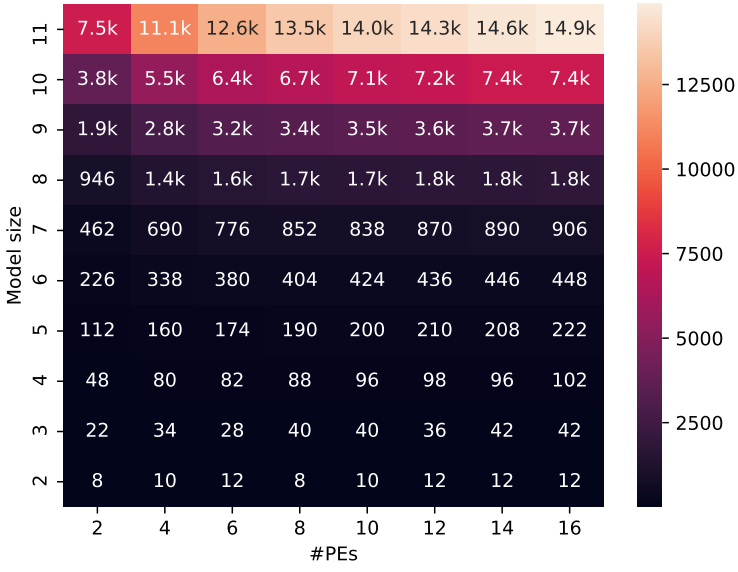


Figure 4.15: *#IOps (reads and writes) for the Weibel model.*

4.8.3 Discussion

From Figure 4.14 it can be seen that the I/O performance is almost completely linear in terms of the number of I/O operations. The graph corresponds to an average read speed of 2.6 Mb/s and write speed of 2.8 Mb/s. The difference in read and write speed can be attributed to the way writes and reads are implemented over PCIe. A read is done via a *read request* which is responded to by a *read completion*. A read is composed of two packets. A write does not involve a response, and is thus only one packet and therefore faster. Any small deviations from the linear trend is likely attributed to variability coming from the operating system.

Linear performance is expected in this case since the communication with the FPGA card is done via a character device, one 32-bit word at a time, and is unbuffered. There are no caches along the way. Furthermore, since the PCIe packets always are the same length due to the constraints in the I/O module described in Section 3.4.4, the overhead imposed by the PCIe protocol is constant, no matter how much data is sent.

From Figure 4.15 we can see that the largest synthesizable model (model size 9 with 16 PEs) has about 4000 I/O operations in the worst case. This means that the transfer time for that configuration is 0.04 seconds. Thus, the transfer time in the worst case is a non negligible portion of the total simulation time.

The I/O performance is also heavily dependant on use case. If the purpose of the simulation is to examine the internal state of a system, the I/O performance can be a limiting factor. If, however the simulation is to be used only as a part in a larger co-simulation, it might not be necessary to expose all internal state, but only a few output variables. This would be a better use case for the Monza system as it is now. The I/O performance can probably be significantly improved if support for sending more than one word at a time was to be implemented.

One way to improve performance would be to implement a better I/O module without restrictions on how large of a payload can be handled per packet. This, combined with necessary modifications to the driver would greatly decrease the transfer time. An improved I/O module would be required anyway if the system's clock frequency is to be decoupled from the PCIe clock.

5

Conclusions

This thesis has produced a system that can automatically tailor a custom hardware architecture to execute a model described by ODE equations on an FPGA. Additionally, the system allows interaction and simulation of models running on the FPGA when used in conjunction with a general purpose computer.

Perhaps most importantly, the system supports models written in – albeit a subset of – Modelica, a versatile programming language used in the industry, that is aimed at modeling complex physical systems. At the time of writing, the authors are not aware of other any system capable of this.

From the results in chapter 4 one can conclude that it is feasible to solve these kind of modeling problems on a FPGA, but that there are lots of future work to be done in order to have an efficient system in the general case. The answers to the research questions stated in Section 1.4 are as follows:

1. Simulations using the proposed system is faster than OMC for models with up to 127 state variables, but slower for larger ones. Likewise, the system is close to OMC in terms of correctness for small models, but diverges as the models grow larger.

2. The time to transfer data to and from the FPGA card is approximately 2.6 Mb/s for reading and 2.8 Mb/s for writes.

3. As the models grow larger, block RAM availability becomes the limiting factor.

4. In the system's current form, the FPU is a major bottleneck.

5.1 Future Work

As stated, a major bottleneck of the current system is the FPU. The currently integrated FPU is a single precision unit which is clearly not enough for very large models. Investigating other FPU designs, including pipelined versions, would therefore be beneficial. However, the use of a piplined FPU would not be trivial, since it would require a more advanced scheduler.

Investigating double-precision FPUs would also be intresting, because it would allow smaller errors. On the other side of the spectrum, the use of fixed-point would also be highly attractive, for example by incorporating the findings of [27]. For the use-cases where the reduced precision is acceptable, these methods could provide great reductions in execution time.

As it is now, the Monza system receives its clock signal from the PCIe connector. This has the benefit of making the system simpler since there are no synchronization issues when communicating with the host computer. However, it also causes the increments in possible clock frequency to be very large since they are set by the PCIe standard. The system can run in three different frequencies, 62.5, 125 and 250 MHz. The latter of the three was found unusable for large models. If the Monza system clock was decoupled (but still synchronized) from the PCIe connector, a more precise measurement of the possible clock frequency could have been done, and possibly resulting in higher performance.

In order for the system to be usable for very large models, the available block RAM must be utilized more efficiently. Figure 4.13 shows that the main problem regarding memory usage is the transfer memory in the interconnect module. This memory quickly becomes very big for larger models. However, the memory rows are very sparsely populated with actual instructions, and there is rarely need to be able to load all registers at once. This memory would therefore be a good candidate for some kind of associative memory, where only rows which actually contain instructions are stored. This was in fact the original idea, but due to time constraints it was not implemented.

Appendix

A

Test Code

Listing A.1: *I/O test program.*

```
import struct, random, time, monza

MAX_ADDR, MAX_PE = 500, 8

def dtof(value):
  return struct.unpack("f", struct.pack("f", value))[0]

def run_test(n_writes):
  writes = []
  for x in range(n_writes):
    pe, addr = x // MAX_ADDR, x % MAX_ADDR
    value = dtof(random.uniform(-4583.0, 2017.3))
    if pe < MAX_PE:
      writes.append((pe, addr, value))

  reads = [[p, a, None] for p, a, _ in writes]

  with open(monza.DEVFILE, "r+b") as dev:
      r1_start = time.time()
      for pe, addr, _ in reads:
          monza.read(dev, pe, addr)
      r1_end = time.time()

      w1_start = time.time()
      for pe, addr, value in writes:
          monza.write(dev, pe, addr, value)
      w1_end = time.time()
```

```
        r2_start = time.time()
        for i, (pe, addr, _) in enumerate(reads):
            reads[i][2] = monza.read(dev, pe, addr)
        r2_end = time.time()

    assert all(r[2] == w[2] for r,w in zip(reads, writes))

    r1_time = r1_end - r1_start
    w1_time = w1_end - w1_start
    r2_time = r2_end - r2_start
    total_time = r2_end - r1_start
    slack_time = total_time - (r1_time + w1_time + r2_time)
    print(",".join(map(str, (r1_time, w1_time, r2_time, total_time
        , slack_time))))

if __name__ == "__main__":
    for n in range(0, 4001, 100):
        for i in range(50):
            print(n, end=",")
            run_test(n)
```

B

Test Models

Listing B.1: *Weibel test model code for model size 6.*

```
model Weibel2
    constant Integer N=6;
    constant Integer n_nodes = IntExp2(N)-1;
    constant Integer n_leaves = IntExp2(N-1);

    function IntExp2
        input Integer x;
        output Integer res;
    algorithm
        res := 1;
        for i in 1:x loop
            res := res*2;
        end for;
    end IntExp2;
    constant Real fm = 1.0;

    constant Real c1 = 2.0;
    constant Real c2 = 3.0;
    constant Real c3 = 4.0;
    constant Real c4 = 5.0;
    constant Real c5 = 6.0;
    constant Real c6 = 7.0;

    Real v[n_nodes];
    Real f[n_nodes-n_leaves];

initial equation
  for i in 1:n_nodes - n_leaves loop
```

```
      f[i] = 0.1*i;
  end for;

  v = {6.786445617675781, 9.37424087524414, 6.808694362640381,
      1.198136806488037, 2.1012654304504395, 2.206273317337036,
      1.5495665073394775, 5.158806800842285, 3.0606167316436768,
       2.950281858444214, 3.431861639022827, 1.3502037525177002,
       4.782289505004883, 8.883207321166992, 5.120411396026611,
      7.627836227416992, 3.891678810119629, 9.913298606872559,
      3.482109785079956, 3.0694785118103027, 6.7850847244262695,
       6.99486780166626, 3.7678937911987305, 4.089155197143555,
      2.1098320484161377, 6.150906085968018, 2.555266857147217,
      5.220844745635986, 0.5021372437477112, 4.816085338592529,
      8.649983406066895, 4.125332832336426, 5.973865985870361,
      1.7416325807571411, 0.15281495451927185,
      2.520205020904541, 1.759538173675537, 8.181405067443848,
      2.7276575565338135, 9.662988662719727, 7.08461856842041,
      8.473209381103516, 1.0261244773864746, 4.06883430480957,
      3.307166337966919, 2.056020736694336, 9.211819648742676,
      7.833808898925781, 1.6420692205429077, 6.026069641113281,
      8.233936309814453, 3.8474819660186768, 8.239087104797363,
      5.348986625671387, 7.36916971206665, 8.126617431640625,
      0.9753096103668213, 7.779897689819336, 4.526229381561279,
      8.444256782531738, 1.3654708862304688, 3.0369861125946045,
       5.347217559814453};
equation
    // volume larynx
    der(v[1]) = fm*c1 + v[1]*c2 + f[1];
    // volume bronchioles
    for n in 1:IntExp2(N-2)-1 loop
        //left child
        der(v[2*n]) = f[n]*c1 + v[2*n]*c2 + f[2*n];
        //right child
        der(v[2*n+1]) = f[n]*c1 + v[2*n+1]*c2 + f[2*n+1];
    end for;

      // volume alveols
  for i in IntExp2(N-1):n_nodes loop
    v[i] = 0.3*i;
  end for;

    // airflow bronchioles
    for n in 1:IntExp2(N-1)-1 loop
        //left and right child
        der(f[n]) = v[n]*c3 - f[n]*c4 - v[2*n+1]*c5 - v[2*n]*c6;
    end for;
end Weibel2;
```

Listing B.2: *Tanks test model code for model size 64.*

```
within SimpleFluid.Examples;

model Tanks
constant Integer N=64;
SimpleFluid.Tank tanks[N](level_init={4.88618803024292,
    5.848886966705322, 6.793025493621826, 4.230380535125732,
    3.683314561843872, 9.884590148925781, 2.6091654300689697,
    7.77100133895874, 4.3122100830078125, 3.5852038860321045,
    0.6385794878005981, 8.63578987121582, 7.020041465759277,
    9.030107498168945, 4.516118049621582, 6.769209861755371,
    1.1891028881072998, 3.9795360565185547, 2.072319746017456,
    0.42101427912712097, 9.479613304138184, 2.1589436531066895,
    1.4635448455810547, 1.9797004461288452, 3.7803196907043457,
    5.463912487030029, 1.5133436918258667, 9.8868989944458,
    9.8298921585083, 1.4840201139450073, 4.05906867980957,
    6.799294948577881, 8.776565551757812, 4.95405912399292,
    9.170466423034668, 3.2246031761169434, 4.984408855438232,
    4.986465930938721, 6.700681686401367, 2.0199131965637207,
    6.097706317901611, 2.1877310276031494, 3.40220308303833,
    9.625664710998535, 8.990080833435059, 8.181183815002441,
    0.3546826243400574, 1.4836688041687012, 2.568819046020508,
    7.841665744781494, 8.423333168029785, 5.829481601715088,
    7.181316375732422, 8.07055377960205, 0.6635913252830505,
    0.8464313745498657, 8.688953399658203, 0.3941583037376404,
    2.250906467437744, 0.4063202738761902, 0.152851402759552,
    8.439546585083008, 3.305943727493286, 1.6069005727767944});
SimpleFluid.Pipe pipes[N-1];

equation
  for i in 1:(N-1) loop
    connect(tanks[i].port, pipes[i].port_a);
    connect(pipes[i].port_b, tanks[i+1].port);
  end for;
end Tanks;
```

Listing B.3: *Subcomponents of Tanks model.*

```
within SimpleFluid;
model Tank
  parameter Real fluid_density = 998.0;
  parameter Modelica.SIunits.Acceleration g = -9.82;
  parameter Modelica.SIunits.Length level_init = 2.0;
  parameter Modelica.SIunits.Area area = 1.0;

  Modelica.SIunits.Length level;
public
  SimpleFluid.Port port;
initial equation
  level = level_init;
```

```
equation
  fluid_density*g*level = port.p;
  area*der(level) = port.q;
end Tank;

model Pipe
  parameter Modelica.SIunits.Length length = 1.0;
  parameter Modelica.SIunits.Area area = 0.05;
  parameter Real resistance = 10000;

  import SimpleFluid.Resistance;
  import SimpleFluid.Inertance;

  Port port_a;
  Port port_b;

  Resistance r;
  Inertance i;
equation
  connect(port_a  , r.port_a);
  connect(r.port_b, i.port_a);
  connect(i.port_b, port_b);
end Pipe;

model Inertance
  parameter Real fluid_density = 998.0;
  parameter Modelica.SIunits.Area area = 0.2;
  parameter Modelica.SIunits.Length length = 0.05;

  SimpleFluid.Port port_a;
  SimpleFluid.Port port_b;

protected
  Modelica.SIunits.Pressure p_diff;
equation

  p_diff = port_b.p-port_a.p;

  p_diff = der(port_a.q)*length*fluid_density/area;

  port_a.q + port_b.q = 0;
end Inertance;

model Resistance
  parameter Real resistance = 500;

  SimpleFluid.Port port_a;
  SimpleFluid.Port port_b;
equation
  port_a.q + port_b.q = 0;
```

```
  port_a.q = (1/resistance)*(port_b.p-port_a.p);
end Resistance;
```

Bibliography

[1] Ingela Lind and Henric Andersson. Model Based Systems Engineering for Aircraft Systems – How does Modelica Based Tools Fit? In *8th International Modelica Conference, March 20th-22nd, Technical Univeristy, Dresden, Germany*. Linköping University Electronic Press, 2011. Cited on pages 1 and 2.

[2] Nvidia. GPU-Accelerated Applications, March 2016. URL `http://images.nvidia.com/content/tesla/pdf/Apps-Catalog-March-2016.pdf`. Cited on page 2.

[3] Kristian Stavåker. *Contributions to Simulation of Modelica Models on Data-Parallel Multi-Core Architectures*. Linköping : Department of Computer and Information Science, Linköping University, 2015. ISBN 978-91-7519-068-6. Cited on pages 2, 7, and 18.

[4] Dean Karnopp, Donald L. Margolis, and Ronald C. Rosenberg. *System Dynamics: Modeling, Simulation, and Control of Mechatronic Systems*. Hoboken : Wiley, 2012. ISBN 978-0-470-88908-4. Cited on pages 7 and 10.

[5] Jan Thompson, Thomas Martinsson, Per-Gunnar Martinsson, and Johan Thompson. *Wahlström & Widstrands matematiklexikon*. Stockholm : Wahlström & Widstrand, 1991. ISBN 978-91-46-15957-5. Cited on pages 7 and 10.

[6] Uri M. Ascher and Linda Ruth Petzold. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. Philadelphia : Society for Industrial and Applied Mathematics, 1998. ISBN 978-0-89871-412-8. Cited on page 7.

[7] Modelica® - A Unified Object-Oriented Language for Systems Modeling Language Specification Version 3.3 Revision 1, July 2014. Cited on page 9.

[8] François E. Cellier and Ernesto Kofman. *Continuous System Simulation*. New York : Springer Science+Business Media, 2006. ISBN 978-0-387-30260-7. Cited on pages 10 and 11.

[9] Peter Fritzson et al. OpenModelica System Documentation Version 2013-01-28 for OpenModelica 1.9.0 Beta3, January 2013. Cited on page 14.

[10] Katherine Compton and Scott Hauck. Reconfigurable Computing: A Survey of Systems and Software. *ACM Comput. Surv.*, 34(2), June 2002. ISSN 0360-0300. doi: 10.1145/508352.508353. Cited on pages 16 and 17.

[11] T. J. Todman, G. A. Constantinides, S. J. E. Wilton, O. Mencer, W. Luk, and P. Y. K. Cheung. Reconfigurable Computing: Architectures and Design Methods. *IEE Proceedings - Computers & Digital Techniques*, 152(2), March 2005. ISSN 13502387. doi: 10.1049/ip-cdt:20045086. Cited on page 17.

[12] Xilinx. Zynq-7000 All Programmable SoC Overview, September 2016. URL `https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf`. Cited on page 17.

[13] Intel. Arria 10 Overview, May 2017. URL `https://www.altera.com/en_US/pdfs/literature/hb/arria-10/a10_overview.pdf`. Cited on page 17.

[14] David F. Bacon, Rodric Rabbah, and Sunil Shukla. FPGA Programming for the Masses. *Communications of the ACM*, 56(4), April 2013. ISSN 00010782. doi: 10.1145/2436256.2436271. Cited on page 17.

[15] Mahder Gebremedhin. *Automatic and Explicit Parallelization Approaches for Mathematical Simulation Models*. PhD thesis, Linköping University Electronic Press, Linköping, 2015. Cited on page 18.

[16] Matthias Korch and Thomas Rauber. Optimizing Locality and Scalability of Embedded Runge–Kutta Solvers using Block-Based Pipelining. *Journal of Parallel and Distributed Computing*, 66(3), March 2006. ISSN 0743-7315. doi: 10.1016/j.jpdc.2005.09.003. Cited on page 18.

[17] Chen Huang, Frank Vahid, and Tony Givargis. A custom FPGA processor for physical model ordinary differential equation solving. *IEEE Embedded Systems Letters*, 3(4), 2011. ISSN 19430663. doi: 10.1109/LES.2011.2170152. Cited on pages 19 and 44.

[18] Chen Huang, Frank Vahid, and Tony Givargis. Automatic Synthesis of Physical System Differential Equation Models to a Custom Network of General Processing Elements on FPGAs. *ACM Trans. Embed. Comput. Syst.*, 13(2), September 2013. ISSN 1539-9087. doi: 10.1145/2514641.2514650. Cited on page 19.

[19] Chen Huang, Bailey Miller, Frank Vahid, and Tony Givargis. Synthesis of Custom Networks of Heterogeneous Processing Elements for Complex Physical System Emulation. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '12, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1426-8. doi: 10.1145/2380445.2380483. Cited on page 19.

[20] Bailey Miller, Frank Vahid, Tony Givargis, and Philip Brisk. Graph-Based Approaches to Placement of Processing Element Networks on FPGAs for Physical Model Simulation. *ACM TRANSACTIONS ON RECONFIGURABLE TECHNOLOGY AND SYSTEMS*, 7(4), January 2015. ISSN 19367406. Cited on page 19.

[21] T. Yu, C. Bradley, and O. Sinnen. ODoST: Automatic Hardware Acceleration for Biomedical Model Integration. *ACM Transactions on Reconfigurable Technology & Systems*, 9(4), August 2016. ISSN 19367406. Cited on page 19.

[22] T. Yu, O. Sinnen, C. Bradley, and J. Oppermann. Performance Optimisation Strategies for Automatically Generated FPGA Accelerators for Biomedical Models. *Concurrency Computation*, 28(5), 2016. ISSN 15320634. doi: 10.1002/cpe.3699. Cited on page 20.

[23] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. Sebastopol, CA : O'Reilly, 2005. ISBN 978-0-596-00590-0. Cited on page 20.

[24] Yu Kwok. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM COMPUTING SURVEYS*, 31(4), December 1999. ISSN 03600300. Cited on page 21.

[25] Alfred V. Aho. *Compilers: Principles, Techniques & Tools*. Boston : Pearson Addison-Wesley, 2007. ISBN 978-0-321-48681-3. Cited on page 36.

[26] Kahan, W. IEEE Standard 754 for Binary Floating-Point Arithmetic. 1996. Cited on page 50.

[27] U Nordström. Automatic implementation and analysis for fixed-point controllers in modelica using dymola. 2012. URL `http://lup.lub.lu.se/student-papers/record/2629522`. Cited on page 64.