High Level Synthesis for Optimising Hybrid Electric Vehicle Fuel Consumption using FPGAs and Dynamic Programming

Frans Skarman
Master of Science Thesis in Computer Engineering

High Level Synthesis for Optimising Hybrid Electric Vehicle Fuel Consumption using FPGAs and Dynamic Programming:

Frans Skarman

LiTH-ISY-EX–19/5253–SE url

Supervisor: Oscar Gustafsson
            ISY, Linköpings universitet
Daniel Jung
            ISY, Linköpings universitet

Examiner: Mattias Krysander
            ISY, Linköpings universitet

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

Copyright © 2019 Frans Skarman
Abstract
The fuel usage of a hybrid electric vehicle can be reduced by strategically combining the usage of the combustion engine with the electric motor. One method to determine an optimal split between the two is to use dynamic programming. However, the amount of computations grows exponentially with the amount of states which makes its usage difficult on sequential hardware. This thesis project explores the usage of FPGAs for speeding up the required computations to possibly allow the optimisation to run in real time in the vehicle. A tool to convert a vehicle model to a hardware description language was developed and evaluated. The current version does not run fast enough to run in real time, but some optimisations which would allow that are proposed.
Acknowledgements

First of all, I would like to thank my supervisors Oscar and Daniel for all the help, interesting discussions, insights and feedback that they have given me while working on this project. I would also like to thank my examiner Mattias for his insights on both this report, and the project as a whole.

As this thesis marks the end of my engineering degree, I would also like to thank all the teachers and staff, both at Linköping University and at the University of Twente for making the past 5 years as great as they have been.

I would also like to thank everyone who has contributed to the open source software I have used in this project, from the operating system to the libraries used in the project itself.

Finally, I would like to extend my deepest thanks to my parents for always showing interest and supporting me throughout my education.
1 Introduction 1
1.1 Motivation 2
1.2 Related work 2
1.3 Aim 3
1.4 Research questions 3
1.5 Delimitations 3

2 Theory 5
2.1 Terminology 6
2.2 Dynamic programming 7
  2.2.1 Forward search 8
  2.2.2 Parallelism 9
2.3 The vehicle model 9
2.4 Expressions, expression trees and expression graphs 11
2.5 Pipelining 12
  2.5.1 Pipeline depth 13
2.6 Hardware for dynamic programming 13
  2.6.1 Execution time 15

3 Method 17
3.1 Conversion into an expression graph 17
3.2 Converting nodes to hardware description language (HDL) 18
3.3 Pipelining 19
3.4 Boolean values and if-expressions 20
3.5 Special nodes 20
  3.5.1 Estimating square roots 20
  3.5.2 Generating lookup tables 21
3.6 Generating HDL modules 22
  3.6.1 Adding pipeline registers 22
  3.6.2 Only generating each node once 22
  3.6.3 Specifying inputs and outputs 22
3.7 Word length selection 22
3.8 Common patterns 24
Introduction

By adapting the driving pattern of a vehicle to the road ahead, the fuel consumption and emissions of the vehicle can be reduced. This is true for both conventional non-hybrid vehicles [6] and especially hybrid electric vehicles (HEVs) where energy can be stored in the battery for future use. Dynamic programming is one method which has been used successfully for calculating an efficient split between the use of the electric motor and combustion engine [14]. However, the runtime of such an algorithm grows exponentially with the number of inputs and state variables which makes its use for real time calculations in an HEV difficult on a general purpose CPU [15].

Many dynamic programming problems have a high degree of parallelism which can be exploited to reduce the computation time required to solve the problem. Field programmable gate arrays (FPGAs) can use pipelining to exploit the parallelism in a problem in order to quickly solve it while efficiently using the available hardware. This thesis will investigate executing the HEV optimisation algorithms on an FPGA in order to run the optimisation in real time while the vehicle is driving. To achieve this, two things have to be done: first, the model of the vehicle drive train must be implemented on an FPGA and second, the model needs to be run by the dynamic programming algorithm, also on the FPGA.

Every vehicle has a unique vehicle model which is generally written in a general purpose programming language by engineers who are not familiar with hardware design. While the model could be manually converted to an HDL, this would take extra time and resources. Instead, this could be done automatically by a tool which converts such a model written in C++ to FPGA hardware with minimal knowledge of hardware design. This tool is the primary focus of this thesis.

This thesis will also explore possible modifications to the models which improve
the runtime and reduce the amount of required hardware when running on an FPGA. This might include things like using fixed point numbers rather than floating point numbers and converting division by constants to multiplications.

1.1 Motivation

Dynamic programming has been used to optimise the driving pattern of both conventional vehicles [6] and HEVs. However, while it can run in real time on sequential hardware in a non-hybrid vehicle, the computational complexity grows exponentially with the amount of control inputs [15] which makes its usage difficult for HEVs.

The structure of the dynamic programming algorithm allows most of the long running computations to be pipelined in order to massively increase the throughput of the system. This should make it possible to run the algorithm in real time on an FPGA which would reduce the emissions and improve the fuel economy of an HEV. The parallelism could also be exploited by other hardware, such as graphics processing units (GPUs), but they consume significantly more power than FPGAs.

While the primary goal of this project is to optimise the execution of dynamic programming optimisation of HEVs, the methods discussed should be applicable in other fields which also utilise dynamic programming and have real time requirements.

1.2 Related work

Y. Hu and P. Georgiou present an FPGA adaptation of a dynamic programming algorithm for genome sequencing [8]. The implementation presented by the paper is very specific to the genome sequencing problem which means that it is not of much use in this project.

Sean O Settle used OpenCL to perform genome sequencing using dynamic programming on FPGAs [16]. While OpenCL allows FPGA programming without hardware design knowledge, it still requires knowledge of parallelism and OpenCL itself. This means that with the method presented, some extra effort would still be required to convert a vehicle model to FPGA hardware.

Several studies have been conducted on paralleling dynamic programming using other hardware. M. Miyazaki and S. Matsumae present a pipelined dynamic programming implementation for GPUs which is able to produce one output result per clock cycle but is limited by memory bandwidth on the GPU [10]. The problem they are solving assumes that the value at position \( i \) depends on the \( k \) previous values which makes it more difficult to parallelise than the problem solved in this project. M. Cruz, P. Tomás and N. Roma present a method for efficiently performing dynamic programming on a very large instruction word processor [3].

In 2015 R. Nane et.al. conducted a survey of the available high level synthesis
tools. The article presents a long list of available tools as well as performance evaluation of a subset of the tools. [12]. However, most of the non-commercial tools presented do not support converting C++ to hardware.

### 1.3 Aim

In order to run the required calculations on an FPGA, two main things have to happen. First, hardware to perform dynamic programming has to be developed. Second, the vehicle drivetrain models have to be implemented in hardware in such a way that the calculations can run in real time and the previously mentioned dynamic programming hardware can execute them. The primary focus of this project was the model conversion. Potential dynamic programming hardware was considered in order to evaluate the theoretical performance of the full system, but it was not implemented. The models are often written in a general purpose language like C, C++ or Matlab which cannot directly be executed on an FPGA. Therefore, a tool to convert these models from C++ to an HDL had to be developed.

Finally, possible modifications to the optimisation problem that improve the execution time or resource usage on an FPGA will be investigated. These modifications may include both changes to the vehicle model such as replacing maps with functions, as well as modifications to the full optimisation problem.

A model of a mild HEV will be used to evaluate the performance of the resulting hardware.

### 1.4 Research questions

1. How can a vehicle drivetrain model automatically be converted to FPGA hardware?

2. Can an FPGA implementation of the HEV optimisation algorithm be run in real time?

3. What modifications can be made to the optimisation problem to improve the calculation speed and resource usage?

4. What impact do those modifications have on the resulting vehicle fuel consumption and FPGA resource usage?

### 1.5 Delimitations

Because HEV models are often trade secrets and in order to limit the scope of the project, only one sample model was considered.

Additionally, making the HDL conversion 100% automatic was not a goal, instead, the conversion should do most of the work that requires knowledge of hardware design. As pre-processing, a software developer will have to make some changes to the software, like replacing types of variables, possibly speci-
fying their bounds, and in some cases replace programming constructs that can
not be automatically converted.

This project will primarily consider non-plug-in HEVs which have an internal
battery that can not be charged from the power grid as that allows for simpler
calculations when determining the end goal state of the vehicle.
HEVs can be split into different categories depending on the configuration of the powertrain. In serial HEVs, the combustion engine drives a generator which provides electricity to the battery and the electric motor that actually drives the wheels. A parallel HEV uses the combustion engine together with the electric motor to drive the wheels but does not use the combustion engine for providing electric power. Finally, a combined HEV uses the combustion engine both for driving the wheels and for electric power generation [15].

HEVs can also be categorised by their degree of hybridisation, how much power the electric motor contributes to the powertrain. Micro hybrids only use the motor for quickly starting the combustion engine but not for providing torque to the vehicle. In a mild hybrid, the electric motor contributes some torque to the vehicle and can use regenerative braking to save energy. Full hybrids contribute more torque to the vehicle which reduces the size of the combustion engine. Finally, plug-in hybrids have a larger battery which is charged from the power grid and they can drive long distances using only electric power[5]. The model used throughout this thesis models a mild HEV.

In general, an HEV has a set of control inputs which includes things like brake force, electric motor torque, as well as combustion engine torque and gear. The HEV also has a set of state variables which includes things like velocity or kinetic energy, battery state of charge, and fuel amount. In each state, varying the control inputs takes the vehicle to a new future state [15].

The set of states over time (or distance) can be seen as points in an $n + 1$ dimensional space where $n$ is the amount of state variables. Figure 2.1 shows an example of this with a single state variable: velocity. With the car in a specific state at time $t$, any of the control inputs will put the car in another state that is
reachable from the current state within the limits of the control inputs at time $t + \Delta t$. This is represented by the blue triangle in Figure 2.1.

The set of possible states is limited by physical characteristics of the vehicle and road. For example, the speed limit constrains the upper and lower bound on the vehicle velocity, the battery state of charge is bounded in order to not damage the battery and the fuel tank can not have a negative amount of fuel. Similarly, the control inputs are bounded, for example, the engine and motor have upper bounds on the amount of torque they can provide.

There are also additional logical bounds on the control inputs, for example, braking when the same braking force can be achieved using regenerative braking is a waste of energy, and so is running the engine at the same time as applying a braking force. By exploiting the latter point, the control inputs for the engine and braking can be combined into one control input which if negative indicates that the vehicle should brake and if positive, indicates that the combustion engine should be used.

### 2.1 Terminology

To make describing different parts of the optimisation problem easier, some terms will be defined here. A graphical representation of these terms is shown in Figure 2.2. The full optimisation problem takes place in a state space consisting of distance, as well as other state variables like velocity and state of charge. A slice of the state space at a specific distance from the start will be refered to as a, time step or just step. In each step, all state variables have some values, a set of values for these variables will be referred to as a state. A set of state variables which includes distance is called a full state. When performing dynamic programming,
Figure 2.2: A graphical representation of the full state space

the optimisation is done “in reverse”, meaning that the cost of the goal state is evaluated first followed by the step before it. In order to avoid confusion, the next time step will always refer to a future state, not necessarily the next step to be evaluated.

2.2 Dynamic programming

The goal of this project was to implement hardware to search the full state space for a set of control inputs that take the car from a starting state to an end state as efficiently as possible. Efficiently here means that a tradeoff is made between fuel consumption and time taken to get to the destination. In order to achieve this, dynamic programming was used.

Dynamic programming solves recursive optimisation problems where subproblems are re-computed multiple times. This is done by computing and storing the solution to each subproblem once and re-using that solution next time the same subproblem is computed [2]. In this case, the optimization problem to be solved is finding a driving pattern which takes the vehicle from an initial state to an end state as efficiently as possible.

In order to solve this problem, the cost of each point in the state space is defined as the cost of the most efficient path from that point to the destination. This is denoted by $v[d, x]$ where $d$ is the distance and $x$ is a vector of state variables.
The most efficient path to the goal can then be computed by

\[ v[d, x] = \min_{i \in \text{states}(d)} (v[d + \Delta d, i] + \text{travelCost}((d, x), i)) \]  

(2.1)

where the \text{travelCost} function denotes the cost of getting from one state to another, and \text{states} is a set of the possible values of the state variables at time \(d\). We also define the cost in the goal node as 0.

Unfortunately, (2.1) can not be computed directly because there is an infinite amount of possible states and \(\Delta d\) can be arbitrarily small. In order to make the computation possible, the state- and input space must be discretised into a fixed amount of points. In this case, distance has 690 discretisation steps, state of charge and velocity have 30 discretisation steps each, the engine and motor torque also have 30 discrete steps each and the gear has 6 separate steps.

Additionally, which states in the next time step can be reached from one state is not known and must be calculated based on the drivetrain model. This calculation is done by iterating through the possible control inputs to work out the next state if that control input is applied. This means that (2.1) can't easily be calculated directly. Instead, the pseudocode shown in Listing 2.1 is used.

**Listing 2.1: Pseudocode showing dynamic programming when reachable states are unknown**

```plaintext
set the cost to go for all states in the last layer to infinity
set the cost to go of the goal node to 0
for each layer \(n\) starting with the last layer {
    for each state in layer \(n\) {
        let lowest_cost = infinity;
        for each input {
            let reached_state = effect of this input in this state
            lowest_cost = min(lowest_cost, reached_state + cost(input))
        }
    }
}
```

The discretised control inputs will generally not result in a discrete future state, which means that the cost of the resulting state must be computed based on the surrounding states using interpolation.

### 2.2.1 Forward search

After the cost to go for each state has been evaluated by the dynamic programming algorithm, the result can be refined by a forward search through the state space based on the vehicle’s present state.

Here, the discretised grid produced by dynamic programming is refined by interpolating between the points in order to reach a more optimal result. This computation is deemed fast enough to run on a general purpose CPU and is therefore not optimised in this project.
2.2.2 Parallelism

In order for an algorithm to be executed efficiently on an FPGA, some degree of parallelism is required. Luckily, the vehicle can only travel forward in time, and can not travel between states in a single time step. Therefore, the cost of each state in a single time step can be computed in parallel. When the cost of every state in a single time step have been computed, the next time step can not easily be processed until all states in the closest future time step are fully calculated.

2.3 The vehicle model

The vehicle model that was used throughout this thesis has two state variables: battery charge and kinetic energy as well as three inputs: combustion engine torque, electric motor torque, and gear. It also has a large amount of configuration parameters which do not change between states or input, but still affect the performance of the car, for example, torque values at a given motor RPM, minimum and maximum state of charge, and gear ratios.

In any given state and set of inputs, the model computes the state that would be reached using those inputs along with the cost of reaching that state. All the details of the model will not be explained in this thesis, instead, an overview of the different computations that are performed will be given. For more details on a similar model, see [9].

Figure 2.3 shows an overview of the components in the vehicle being simulated.

![Overview of the modelled powertrain](image)

*Figure 2.3: Overview of the modelled powertrain*

While fuel usage is important, if it is the only thing being optimised, the best strategy is not to drive at all. Therefore, a tradeoff between fuel usage and travel time is made, and the cost of a journey is defined as

\[
m_{fuel} \cdot \gamma + T \cdot (1 - \gamma)
\]

where \(m_{fuel}\) is the amount of fuel consumed and \(T\) is the travel time. Gamma is a tunable parameter which controls how much to prioritise fuel usage over arrival
time. If it is 0, only fuel usage will be considered, and if it is 1, only arrival time will be taken into account.

If time is used as a state variable, the cost of each state can be computed as

\[ c_k = \dot{m}_{\text{fuel}} \cdot \Delta t \cdot \gamma + (1 - \gamma) \cdot \Delta t \]  

(2.2)

where \( \dot{m}_{\text{fuel}} \) is the fuel usage per second and \( \Delta t \) is the length of the time step. However, distance is used as a state variable rather than time, which means that the fuel usage calculation is slightly more complex.

The time used in a step \( k \) is computed based on the velocity in the step \( v_{\text{avg},k} \) and step distance \( \Delta d \) as

\[ \Delta t_k = \frac{\Delta d}{v_{\text{avg},k}} \]

This can then be combined with (2.2) to calculate the cost of a set of state variables as

\[ c_k = \frac{\Delta d}{v_{\text{avg},k}} \cdot (\gamma \cdot \dot{m}_{\text{fuel},k} + (1 - \gamma)) \]

This equation shows examples of a lot of the operations that make up the model. \( \gamma \) and \( \Delta d \) are constants which are used as operands to arithmetic operations. \( \dot{m}_{\text{fuel}} \) and \( v_{\text{avg},k} \) are non-constant variables which are also arithmetic operands.

The current consumed by the electric motor in step \( k \) is computed by

\[ I_{\text{bsg},k} = \frac{V_{\text{oc}}(\text{SOC}) - \sqrt{V_{\text{oc}}(\text{SOC})^2 - 4 \cdot R_0 \cdot P_{\text{bsg},k}}}{2 \cdot R_0(\text{SOC})} \]

where \( V_{\text{oc}} \) and \( R_0 \) is the open circuit voltage and internal resistance of the battery while \( P_{\text{bsg},k} \) is the desired power from the electric motor at step \( k \). This equation showcases the use of both the square, and square root functions as well as more arithmetic operations. Additionally, \( V_{\text{oc}}(\text{SOC}) \) and \( R_0(\text{SOC}) \) are functions of the current state of charge. In this model, \( V_{\text{oc}}(\text{SOC}) \) and \( R_0(\text{SOC}) \) are computed by interpolation between values in a lookup table. The model contains interpolation of both one dimensional and two dimensional functions.

Another interesting computation is the calculation of the torque at the wheels which is calculated by
\[ T_{\text{wheel}} = \begin{cases} \alpha \eta T_{pt} & \text{if } T_{pt} \geq 0 \\ \alpha \cdot T_{pt} & \text{otherwise} \end{cases} \]

where \( T_{pt} \) is the torque input to the gearbox, \( \alpha \) is the ratio between motor rotation and wheel rotation and \( \eta \) is the efficiency of the gearbox. In order to compute this, logic to handle conditionals is required in addition to arithmetic operations.

### 2.4 Expressions, expression trees and expression graphs

An expression is a piece of a program which calculates a value when executed [7]. The model used throughout this project produces several values, each of which can be computed by a single expression. Expressions can be represented by expression trees which allows simple conversion to and from the expression written in a programming language.

In an expression tree, each node represents a value which is either a leaf node in the tree, or the result of another expression. A directed edge from one node to another indicates that the first node is an input operand to the other. Figure 2.4 shows an example of an expression tree which represents the expression \((a + b) \cdot c\).

![Figure 2.4: An expression tree which represents the expression \((a + b) \cdot c\)](image)

An expression is a piece of a program which calculates a value when executed [7]. The model used throughout this project produces several values, each of which can be computed by a single expression. Expressions can be represented by expression trees which allows simple conversion to and from the expression written in a programming language.

In an expression tree, each node represents a value which is either a leaf node in the tree, or the result of another expression. A directed edge from one node to another indicates that the first node is an input operand to the other. Figure 2.4 shows an example of an expression tree which represents the expression \((a + b) \cdot c\).

The model used throughout this project computes three values as explained in Section 2.3. These values can be represented as three separate expressions and therefore three expression trees. However, variables and re-used values can not be represented in an expression tree. To see why, consider how the following code would be represented:

```c
\text{c} = \text{a} + \text{b};
\text{d} = \text{a} + \text{c};
```

Here, \( \text{d} \) can be represented as a single expression by expanding it to \( \text{d} = \text{a} + \text{a} + \text{c} \) which can then be converted to an expression tree. However, this removes the information that the two \( \text{a} \) values are the same.

Therefore, instead of an expression tree, an extension of an expression tree, which for lack of a better term will be called an expression graph is used. Unlike an
expression tree, an expression graph allows a node to be the operand of multiple nodes. This means that an expression graph is a directed acyclic graph rather than a tree which can be converted into an expression tree through depth-first traversal. Figure 2.5 shows an example of an expression tree on the left and an equivalent expression graph on the right.

![Figure 2.5: An example of the difference between an expression tree (left) and expression graph (right). Both represent the calculation of $c = a + b$; $d = a + c$.](image)

### 2.5 Pipelining

When designing hardware for a large expression, the straightforward solution of computing the whole expression using a single large combinatorial circuit is often very inefficient. As an example, consider the expression $a + b + c$ which could be implemented using two adders as shown in Figure 2.6.

![Figure 2.6: Graph representing a circuit for calculating $(a + b) + c$.](image)

In this case, the result of the second addition depends on the result of the first, which means that the propagation delay, the delay between input and output, of the circuit is the sum of the propagation delays of each adder. Additionally, once the result has been computed by the first adder, that adder is unused until the rest of the circuit is done with its calculation. A common way to work around this and provide more efficient hardware usage is to use pipelining where computations for a future result are started before the previous result is done. For example, if $a + b + c$ is to be computed for a large amount of values, $a_2 + b_2$ can be calculated at the same time as $(a_1 + b_1) + c_1$. This is achieved by adding registers after each calculation as shown in Figure 2.7.

This does not speed up the individual computations, in fact they generally become slower when pipelining because the time used in each stage has to be uniform which means that the whole pipeline is limited by the slowest step. How-
ever, pipelining increases throughput as one new computation can be started every clock cycle.

### 2.5.1 Pipeline depth

One issue which remains with the circuit shown in Figure 2.7 is that the value of \((a_1 + b_1)\) will be added to \(c_2\) rather than \(c_1\) as one would expect. In order to fix that, a delay register must be added between \(c\) and the addition node.

In order to calculate the required delay between an output and an input of an expression, the depth of a node is defined as the earliest time at which the value of the node can be computed. The value of nodes with no inputs can be “computed” at any time, and as such, their depth is 0. The value of all other nodes can be computed as soon as the output of all preceding nodes is available. The depth \(d_n\) of node \(n\) can then be calculated by the following equation where \(t_i\) is the computation time of node \(i\).

\[
  d_n = \max_{i \in \text{operands}(n)} (d_i + t_i)
\]  

(2.3)

When the depth is known, the required delay slots between a node and one of its operands can be calculated as \(d_n - d_i - 1\) where \(d_n\) is the depth of the node and \(d_i\) is the depth of the operand.

### 2.6 Hardware for dynamic programming

In this section, the structure of the hardware that could be used to execute the model and perform dynamic programming is presented along with the theoretical performance characteristics that impact the usefulness of the generated model hardware.

The algorithm which is to be performed by the hardware the one listed in Listing 2.1. This is repeated below for reference:

```plaintext
set the cost to go for all states in the the last layer to infinity
set the cost to go of the goal node to 0
for each layer \(n\) starting with the last layer {
    for each state in layer \(n\) {
        let lowest_cost = infinity;
        ```
for each input {
  let reached_state = effect of this input in this state
  lowest_cost = min(lowest_cost, reached_state + cost(input))
}

Figure 2.8: Overview of the dynamic programming hardware

An overview of the hardware is shown in Figure 2.8. This The middle for-loop is emulated by the state_iterator module which iterates through all the available states in the model. In each state, all possible inputs are iterated through by the input_generator module which emulates the innermost for-loop.

The inputs, along with the state are passed to the model which computes the resulting state from those inputs, along with the cost of those inputs which is then passed to a RAM module where the cost of previous states that are closest to the reached state are stored. The cost of the reached state is calculated by interpolating those values, and the result together with the cost of the inputs makes up the total cost of travelling from the current state to the end state, starting with the current inputs.
This cost is then passed to the minimizer module which selects the minimum cost from all inputs in this state. Once all the inputs for one state have been tested, the state_summarizer module writes the cost of the state to RAM and sends a signal to the state_iterator to increment the state to check.

The model execution in a single state is pipelined which means that the input generator sends one input to the model every clock cycle. Once each input in a single state has been evaluated, the pipeline can either be flushed before starting the next state, or the pipeline can start calculating the cost of the next state right away. In the second case, the pipeline still has to be flushed when changing between time steps because the output of the model depends on the cost of states in the previous time step.

### 2.6.1 Execution time

The amount of steps required to solve an optimisation problem with the above hardware depends on the size of the optimisation problem. Consider a problem with $t$ time steps, $n$ possible states, and $i$ inputs where the hardware to calculate the cost of a single input has a pipeline depth of $d$.

As mentioned before, one input can be evaluated per clock cycle in a single state which means that the time required to compute the cost for $k$ inputs is $k + d$.

In the case where the pipeline is flushed after each state, this means that the amount of clock cycles required to complete the optimisation is given by following equation:

$$ T = tn \cdot (i + d) $$

Similarly, if the pipeline is only flushed after each time step, the amount of clock cycles required is given by the following equation:

$$ T = t \cdot (ni + d) $$

Thus, if the pipeline depth is small compared to the amount of inputs, its impact is negligible in both cases.
In this chapter, the C++ to HDL converter is described, along with the methodology used to evaluate the performance of the hardware.

The source code of the HDL converter is available at https://gitlab.com/TheZoq2/thesis/tree/master/hdlconverter

### 3.1 Conversion into an expression graph

In order to simplify the process of converting C++ expressions into an HDL, the expressions are first converted into an expression graph. In order to achieve this, all variables and calculations in the expressions are replaced by a custom type which represents a node in the tree. The type keeps track of which operation is to be performed as well as which operands will be used for the operation. Listing 3.1 shows an example of an abstract class for such a node. The function `operands` returns all the operands which are inputs to this node.

**Listing 3.1: Example of an expression tree node**

```cpp
class Node {
public:
    vector<Node*> operands() const = 0;
};
```

This interface is then implemented for each operation that is used in the model. Listing 3.2 shows an example of a node for performing addition as well as a node which represents a constant.

**Listing 3.2: Example of an addition and a constant node**

```cpp
class Add : public Node {
    public:
```
Add(Node* lhs, Node* rhs) : lhs(lhs), rhs(rhs) 
{
vector<Node*> operands() const override {
    return {lhs, rhs};
}
private:
    Node* lhs;
    Node* rhs;
};

class Constant : public Node {
public:
    Constant(double value) : value(value) {}
    vector<Node*> operands() const override {
        // A constant does not require any calculation
        // which means that it has no operands
        return {};
    }
private:
    double value;
};

In order to make usage of this API more convenient, custom operators are also defined for the Node class which allows them to be used as drop-in-replacements for normal C++ expressions as shown in Listing 3.3. Further, in order to simplify memory management and the definition of the overloaded operators, all nodes are wrapped in shared pointers.

**Listing 3.3: Example of definition and usage of the addition operator for nodes**

```
shared_ptr<Node> operator+(const shared_ptr<Node> lhs, const shared_ptr<Node> rhs) {
    return make_shared(new Add(&lhs, &rhs));
}
```

```
// Example usage
shared_ptr<Node> sum_constants() {
    shared_ptr<Constant> a = make_shared(Constant(5));
    shared_ptr<Constant> b = make_shared(Constant(3));
    return a + b;
}
```

### 3.2 Converting nodes to HDL

In order to convert an expression from the expression graph representation into a HDL, two functions were added to the Node interface: `variable_name` and `get_code`. The former simply returns a string which will be the name of the HDL variable where the result of the calculation associated with the node will be stored. The exact names returned by this function are not important, they could be chosen at random or generated based on the inputs to the node as long as they are always unique and constant for each node.

The `get_code` function returns HDL code which creates the corresponding variable
and gives it its value. Listing 3.4 shows an example of such a function which generates verilog code for an addition node. The size of the resulting register is left out for now but will be discussed in Section 3.7.

Listing 3.4: Function for generating HDL code for an addition node

```cpp
vector<string> Add::get_code() {
    return {
        "wire[..]" + this->variable_name() + ";",
        "assign " + this->variable_name()
        + " = " + this->lhs->variable_name()
        + " + " + this->rhs->variable_name(),
    }
}
```

### 3.3 Pipelining

As explained in Section 2.5, pipelining can be used to increase the throughput of calculated values without adding more computation units but it does require adding extra delay registers when two operands are not available simultaneously.

As was mentioned in Section 2.5.1, the depth of a pipeline node is 0 for all nodes which have no operands, and

\[
d_n = \max_{i \in \text{operands}(n)} (d_i + t_i)
\]

when the depth is known. The computation time of node \(i\) in clock cycles is denoted by \(t_i\). The required delay slots between a node and one of its operands can be calculated as \(d_n - d_i - 1\) where \(d_n\) is the depth of the node and \(d_i\) is the depth of the operand.

The code in Listing 3.4 can then be modified to use the pipeline registers as shown in Listing 3.5. Where the `register_name(n)` function just returns the nth delay register of the node.

Listing 3.5: Listing 3.4 modified to use pipeline registers

```cpp
vector<string> Add::get_code() {
    return {
        "reg[..]" + this->variable_name() + "i;",
        "assign " + this->variable_name()
        + " = " + lhs->register_name(this->depth() - lhs->depth()-1)
        + " + " + rhs->register_name(this->depth() - rhs->depth()-1)
    }
}
```

### Constants

Some nodes, primarily constants do not need pipeline registers as their values never change between iterations. For these nodes, the `register_name` function just returns the default name, and the function that generates pipeline code is overridden to return an empty string.
3.4 Boolean values and if-expressions

So far, the discussion has been centered around nodes with numerical values. However, most programs also contain control logic which requires boolean values and expressions. In order to accommodate boolean nodes, operations that are specific to numerical nodes, such as calculate_value were moved into a subclass of nodes, and boolean nodes were added as a separate subclass with similar functionality.

Most algorithms require some choice based on a boolean value which is most easily done through if-statements. However, converting a C++ if-statement to HDL code from inside C++ is difficult, if not impossible. Therefore, a special node representing an if-expression was introduced.

Like an if-statement, the behaviour of an if-expression depends on a condition, however, unlike an if-statement which performs different computations based on the condition, an if-expression selects between two values depending on the condition.

The if expression node has three operands: the condition, the node containing the value to select if the condition is true and the node to select if the value is false. This means that both the true and false branch are computed and based on the value of the condition node, the true or false value is selected as the value of the node.

3.5 Special nodes

Most nodes that were implemented in this project perform standard arithmetic calculations and were mapped directly to the corresponding verilog version. However, some nodes required extra thought and are documented in this section.

3.5.1 Estimating square roots

The square root of a fixed point value can not easily be calculated in hardware. Instead, it was estimated using the method described in this section.

The square root estimation is based on the following equation:

$$\sqrt{x} = \sqrt{\frac{2^n x}{2^n}} = \sqrt{2^n} \sqrt{\frac{x}{2^n}} = 2^{\frac{n}{2}} \sqrt{\frac{x}{2^n}}$$

If $a$ is within a known bound, $\sqrt{a}$ can be estimated using a lookup table. This means that $\sqrt{\frac{x}{2^n}}$ can be approximated if $n$ is chosen such that $a \leq \frac{x}{2^n} \leq b$. This can be achieved by selecting $n$ as $n = k - m$ where $k$ is the position of the most significant 1 in $x$ and where $m$ is the amount of fractional bits in $x$. This bounds the value of $\frac{x}{2^n}$ between 0 and 1.

If $n$ is an integer, $\frac{x}{2^n}$ can easily be computed by shifting $x$ $n$ steps to the right. Similarly, if $n$ is an even integer, $2^{\frac{n}{2}} \cdot a$ can easily be computed by shifting $a$ $\frac{n}{2}$
steps to the left. However, this does not work when \( n \) is odd. This in turn can be fixed by using \( m = 2n \) as follows:

\[
\sqrt{x} = 2^m \sqrt{\frac{x}{2^m}}
\]

which bounds the value if \( \frac{x}{2^m} \) between 0 and 2 and ensures that \( m \) is always even. This can efficiently be computed in hardware by calculating \( \frac{x}{2^m} \) using bit shifts, using the result to look up the square root, and then shifting the result back \( \frac{m}{2} \) steps.

### 3.5.2 Generating lookup tables

Another common expression which needed extra thought was lookup tables. Lookup tables currently only support integer inputs which are directly mapped to an output fractional value. They were implemented as individual verilog modules in order to make re-use of their values in multiple places easier.

Lookup tables were implemented as two separate classes, one which contains the value mapping and is used to generate the verilog module. The second class is a node subclass which implementation is shown in Listing 3.6.

#### Listing 3.6: Lookup table node class

```cpp
class LutResult : FracNode {
public:
    LutResult(FracNode input, LookupTable lut) : input(input), lut(lut) {}

    get_code() {
        return {
            // "Storage" for the LUT output
            "wire[..] + this->variable_name() + ";",
            // Instantiate lut module
            lut->name " " + this->variable_name + "_lut"
            // Configure the size of the input variable
            + "#(.input_size(" + this->integer_bit_count() + "))"
            // Bind normal variables
            + "(.clk(clk),"
            + " .input(" + input.variable_name() + "),"
            + " .output(" + this->variable_name() "))"
        };

    private:
        FracNode input;
        LookupTable lut;
};
```

The lookup table module itself consists of a register for storing the result, along with a large case statement which assigns the correct value to that result. Initially, this case statement contained the full fractional values with a default branch to catch any non-integer values. However, this caused the synthesis tool to generate multiplexers instead of block RAMs. To avoid this problem, the fractional bits are truncated before being passed to the case statement.
3.6 Generating HDL modules

The discussion so far has been focused on generating code for individual computations, but in order to output useful HDL code, a few more things have to happen. The delay registers for each node have to be added, code for each node has to be generated exactly once, and the inputs and outputs for the module have to be specified.

3.6.1 Adding pipeline registers

Each node can be used multiple times, and the amount of pipeline registers to add depends on the nodes for which that node is an operand. In order to determine the amount of required pipeline registers for each node, each node computes the amount of delay required for each of its operands. The required delay is passed to the operand node which stores the largest amount of delay it receives. When generating code, that amount of pipeline registers is generated.

3.6.2 Only generating each node once

The code for each node should only be inserted into the resulting module once, but since each node can be the operand of multiple other nodes, simply traversing the graph and inserting the code for each node will not work. Instead, the graph is traversed once in a depth first manner and each node gets added to a vector if it is not already in there. Once the vector has been created, the code for each node is generated and inserted in the order that it appears in the vector.

3.6.3 Specifying inputs and outputs

In order to generate a complete and working verilog module, the inputs and outputs to that model must be specified. The model which was used has several parallel outputs for both cost and new states, so multiple outputs must be supported. Luckily, the output nodes are the root nodes of their expression trees which means that returning them suffices to output all required information. Then, the code which generates the model can simply go through all the output nodes and mark them as outputs.

Inputs were slightly more tricky as they are embedded in the expression tree as leaf nodes along with constants which makes it non-trivial to find them. They could have been specified as a list to the function which generates the code, but that would have required extra effort for the user. Instead, the tree is traversed, and all nodes with the input class are added to the module header.

3.7 Word length selection

The value computed in each node must be stored in registers and propagated through the circuit, and because most values in the model are fractional, this could not be done at full precision. Instead, the word length of each value must be selected to produce results that are good enough while not using too much hardware.

All values were stored using a fix point representation since floating point values use much more resources. A fix point number consists of an integer part and a fractional part where the fractional part influences the precision of the stored
number while the integer part controls the minimum and maximum value that can be stored.

In this project, some attempts were made to optimize the size of the fractional part to achieve results with a specified precision. However, those attempts were unsuccessful and will be discussed in Section A. Instead of optimizing the fractional word length individually, the same amount of fractional bits was used for all values in the model, and the amount of fractional bits required was determined by simulation.

Unlike the fractional word length, the integer word length was determined per variable in order to guarantee that no overflow would occur while keeping resource usage low. This was done by calculating the minimum and maximum value of each variable and selecting the smallest integer word length that would fit all those values.

Like many other computations done on the expression graph, the minimum and maximum value in each node can be calculated recursively. To do so, the bounds on values without operands must first be known beforehand. The bounds on constants are trivial to determine, it is simply the constant value, while the bounds on input variables must be specified by a designer.

Then, the bounds on other nodes can be determined recursively from the operation performed in the node and the bounds on the operands. For example, the maximum value of an addition node is the sum of the maximum values of the operands.

This works well for most basic operations, but in some cases, it gives a very pessimistic estimate which can then propagate further through the design. To see why this is a problem, consider the code snippet shown in Listing 3.7.

**Listing 3.7: Example of an expression tree node**

```c
const MIN_VALUE = 0.3;
const MAX_VALUE = 1;
// Holds a value between 0 and 100
Input<0, 100> some_value;

// Cap that value between 0.3 and 1
auto value_capped = _if(some_value < MIN_VALUE, MIN_VALUE,
            _if(some_value > MAX_VALUE, MAX_VALUE, some_value),
    );
auto result1 = value_capped + 1;
auto result2 = 1 / value_capped;
```

In this snippet, a large value called `some_value` is capped from above and below before being used to calculate `result1` and `result2`. It is clear when reading the code that the capped value will be bounded between 0.3 and 1, but there is no general way to determine the effect on variable bounds of an arbitrary if-statement.
Therefore, the program will still think that the capped value is between 0 and 100.

This leads to result1 being stored with way more bytes than is required which is inefficient. However, it leads to much worse problems when calculating the bounds on result2 where the calculation of the maximum value will attempt to divide by zero.

In order to solve this, an extra function was added to if-statements which allows a programmer to manually override the bounds on the node. While this works, it does introduce potential overflow errors if a programmer isn't careful when specifying the bounds. In order to avoid that problem, additional functions were added for common operations which use if-statements to cap variables. These functions return if-nodes with the bounds already applied.

### 3.8 Common patterns

While the HDL converter can convert basic C++ code to verilog, some programming constructs can not currently be directly converted and must be worked around manually. This section presents examples of this, along with workarounds for those problems.

#### 3.8.1 Early returns

One very common pattern in the example model is one where a value is calculated and compared with some minimum or maximum value. If the calculated value exceeds the bounds, the specified input is infeasible and model evaluation is aborted. In the original code, this was implemented using early returns, as shown below.

```c++
v_veh_avg = ( sqrt(x1) + v )/2;
// Ignore really slow average speeds
if(v_veh_avg < 0.01) return -1;
```

In the HDL converter, this leads to a few problems. First, the C++ function has to return the whole expression containing all branches which means that an early return is not possible. Second, an early return would not be easily implemented in a pipeline. In order to work around that, a boolean flag called failed was added. Each time an early return would have happen in C++, that flag is updated to hold or(old_value, condition). If this flag is true at the end of the computation, the calculated values can be discarded.

However, the early returns also often prevent future calculations involving out of bounds values, which with this system propagate through to those calculations. This problem is similar to the out of bounds values that occur when if-statements are used to cap values. However, in this case, since there is no early return node, the programmer manually has to insert additional code to cap the values.

#### 3.8.2 Interpolation

As mentioned in Section 2.3, the model contains several instances of functions being approximated by interpolation of values from a lookup table. The interpo-
lation consists of regular arithmetic operations with some array lookups. As all those operations were already implemented in the HDL converter and the interpolation functions were already written in C++, a special interpolation node is not used.

### 3.9 Simulating the expression tree

In order to enable testing of various things like bit lengths of variables, and to ease debugging of the generated expression graph, some functions were added to execute the expression graph on a normal CPU.

In order to do so, a function to calculate the value of a node was added, this function recursively calls itself on all the operands before performing some local computation based on the operand values. An example of such a function for addition nodes is shown in Listing 3.8 where `ExecutionType` is the type used to represent values. When verifying the translation of a model into an expression tree, this can be set to the same type as what is used in the original model. On the other hand, when evaluating various data types for hardware implementation, it can be changed to whichever type is being evaluated.

**Listing 3.8: Example of the calculate_value for an addition node**

```cpp
ExecutionType Add::calculate_value() {
    lhs_value = this->lhs->calculate_value();
    rhs_value = this->rhs->calculate_value();
    return lhs_value + rhs_value;
}
```

When simulating using fix point values, the `FpF` class from the `MFixedPoint` library was used with 128 bit integers as the base value.

#### 3.9.1 Performance issues and caching

The above methods work well for small expressions, but as the size of the expression grows, a lot of nodes will be recomputed multiple times which in the case of the sample model lead to infeasible computation times. To see why, consider the graph shown in Figure 3.1.

Here, the value of node $a$ will be computed once when calculating $b_1$, once when calculating $b_2$ and once when calculating $b_3$. For larger graphs with more branches, this leads to a lot of recomputations. The sample vehicle model has roughly 500 unique nodes but in total, those nodes get visited over 100 000 times when traversing the tree recursively.

In order to solve this problem, the `calculate_value` function was extended with a cache. If the value of this node has been calculated before, the old value is returned, otherwise it is recomputed. Additionally, a function to clear the cache was added. The implementation can be seen in Listing 3.9.

**Listing 3.9: Node class augmented with a cache for runtime values**

---

1 [https://github.com/gbmhunter/MFixedPoint](https://github.com/gbmhunter/MFixedPoint)
3 Method

```cpp
class Node {
public:
  // Function that actually recomputes the value.
  // Replaces calculate_value in subclasses
  virtual ExecutionType recalculate_value() = 0;

  ExecutionType calculate_value() {
    if(!cached_value.has_value()) {
      cached_value = this->recalculate_value();
    }
    return cached_value.value();
  }

  void mark_dirty() {
    this->cached_value = std::nullopt;
  }

private:
  std::optional<ExecutionType> cached_value;
};
```

The downside to this approach is that the cache has to be cleared when input values change which itself requires traversal of the whole tree. The solution to this was to flatten the expression graph after it is generated by recursively traversing it and inserting all unique nodes into a set data structure. This set can then be iterated over in order to reset the cache for each node.

### 3.9.2 Debugging and debugging tools

A result of the described structure when simulating expression execution is that the execution does not happen in the source code of the function being executed, it happens when the `calculate_value` function is executed. This makes debugging difficult as print statements or debugger breakpoints can’t be placed between calculations, only inside the `calculate_value` function for a specific node class.

In order to aid debugging, a method to output the expression graph of all the ancestors of a node was added. This method outputs the name of a node, along

**Figure 3.1:** Example of expression graph where one node is recomputed multiple times when calculating node values recursively
with values from the node in a graphviz\textsuperscript{2} graph.

3.10 Performance evaluation

In this section, the methods used for evaluating the performance of the HDL converter will be presented.

When the output quality is determined, the original CPU version of the model using double precision floating point values is used as ground truth.

3.10.1 Synthesis

In order to evaluate the maximum clock frequency of the design, as well as the amount of FPGA resources it uses, the model was synthesised using tools included in Xilinx ISE 14.7 which were executed using a makefile written by GitHub user duskwuff\textsuperscript{3}. The target hardware was set to xc6vlx760-FF1760. The hardware usage was recorded after the synthesis step while the maximum frequency was recorded after the place and route step.

The bulk of FPGA hardware consists of a lookup table paired with a flip-flop which makes the number of such pairs used by a design a useful measure of FPGA resource usage. Additionally, when large chunks of data is to be stored, discrete block RAMs are used. These are not constructed from LUT-flip-flops which means that the amount of RAMs used is reported separately.

3.10.2 Output quality

As all values are represented as fixpoint numbers with the same amount of fractional bits, the quality of the output depends on the amount of bits used. The simulation process described in Section 3.9 was used to determine the quality as a function of the number of fractional bits.

The simulation was run on two different road profiles which both contain a mix of urban and highway driving. The first is the same profile as was used by Olin et.al. in [13]. The second profile was generated for this project. For convenience, they will be referred to as the “American” and “Swedish” road profile because of the speed limits used.

In order to determine where to start searching for the required amount of fractional bits, the original model was first executed using double precision floats with truncation to a certain amount of fractional bits at the output.

\textsuperscript{2}https://graphviz.org/
\textsuperscript{3}https://github.com/duskwuff/Xilinx-ISE-Makefile
This chapter presents the results of the evaluation described in Section 3.10. Section 4.1 presents the model output quality at different amounts of fractional bits. In Section 4.2 the resource usage and performance of the model as reported by the synthesis tool will be presented. This is further analysed in Section 4.3 to determine the theoretical running time of a full optimisation. Finally, in Section 4.4, various model changes and their effects on execution time and resource usage will be presented.

### 4.1 Word length effect on output

Figure 4.1 shows the fuel consumption achieved when running the model using double precision floats and truncating the output to various bit lengths. From this simulation, it was determined that anything less than 12 fractional bits would introduce large errors into the output.

Figure 4.2 shows the total cost over a test sequence as a function of the amount of fractional bits used for all the variables in the computation. These results seem to indicate that 20 bits is enough to achieve a result that is within 1% of the original model, while 28 bits is required for output that is near identical to the original model.
Figure 4.1: Model output as a function of the amount of fractional bits which the minimum cost in each state was truncated to. All other values were computed at full precision. The red line represents the output with no truncation.

Figure 4.2: Model output as a function of the amount of fractional bit used to store every variable. The blue plot is the result of American the road profile while the red is the Swedish road profile. The dashed lines represent the output of the original CPU implementation.

**Velocity profile**

Figure 4.3 and 4.4 show the optimal velocity and state of charge in each state as calculated by the model when run at different word lengths. This shows, as one
might expect, that the driving pattern is more different from the original model at 12 bits than at 20 bits. However, it still differs slightly, even at 20 bits.

Figure 4.5 shows the optimal velocity as calculated at 20, 25 and 31 bits. It shows that the velocity profile is identical to the original at 31 bits, and that there is no difference between the results when running with 20 or 25 bits.

**Figure 4.3:** Optimal velocity on the American road profile as calculated by the model when executed with varying word lengths

**Figure 4.4:** Optimal state of charge on the American road profile as calculated by the model when executed with varying word lengths
4.2 Synthesis results

Table 4.1 contains the amount of FPGA resources used by the model as well as the minimum clock period at which the model can run. These numbers are also shown as graphs in Figures 4.6, 4.7 and 4.8. It looks like all three resource usage metrics increase linearly with the amount of fractional bits. However, the minimum clock period has quite a bit of noise. This might be caused by the stochastic nature of HDL synthesis.

**Table 4.1: Synthesis results**

<table>
<thead>
<tr>
<th>Fractional bits</th>
<th>LUT flip-flop pairs</th>
<th>Min. clock period (ns)</th>
<th>Block RAMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>71 992</td>
<td>92.105</td>
<td>61</td>
</tr>
<tr>
<td>13</td>
<td>75 151</td>
<td>90.137</td>
<td>67</td>
</tr>
<tr>
<td>14</td>
<td>79 751</td>
<td>100.529</td>
<td>71</td>
</tr>
<tr>
<td>15</td>
<td>82 090</td>
<td>105.483</td>
<td>75</td>
</tr>
<tr>
<td>16</td>
<td>84 847</td>
<td>105.621</td>
<td>79</td>
</tr>
<tr>
<td>17</td>
<td>90 123</td>
<td>110.544</td>
<td>83</td>
</tr>
<tr>
<td>18</td>
<td>93 012</td>
<td>114.141</td>
<td>87</td>
</tr>
<tr>
<td>19</td>
<td>97 724</td>
<td>112.918</td>
<td>91</td>
</tr>
<tr>
<td>20</td>
<td>101 849</td>
<td>118.924</td>
<td>95</td>
</tr>
<tr>
<td>21</td>
<td>105 156</td>
<td>123.494</td>
<td>99</td>
</tr>
<tr>
<td>22</td>
<td>108 868</td>
<td>132.804</td>
<td>105</td>
</tr>
<tr>
<td>23</td>
<td>113 112</td>
<td>151.650</td>
<td>109</td>
</tr>
<tr>
<td>24</td>
<td>118 441</td>
<td>142.609</td>
<td>113</td>
</tr>
<tr>
<td>25</td>
<td>123 025</td>
<td>154.799</td>
<td>118</td>
</tr>
<tr>
<td>26</td>
<td>128 584</td>
<td>161.420</td>
<td>122</td>
</tr>
<tr>
<td>27</td>
<td>133 548</td>
<td>159.873</td>
<td>125</td>
</tr>
<tr>
<td>28</td>
<td>133 747</td>
<td>161.874</td>
<td>129</td>
</tr>
</tbody>
</table>

*Figure 4.5: Optimal velocity on the american road profile at higher bit counts.*
4.2 Synthesis results

<table>
<thead>
<tr>
<th>Fractional Bits</th>
<th>LUT Flip-Flop Pairs</th>
<th>LUT Flip-Flop Pairs</th>
<th>Block RAMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>139 403</td>
<td>168.205</td>
<td>133</td>
</tr>
<tr>
<td>30</td>
<td>155 937</td>
<td>170.874</td>
<td>138</td>
</tr>
<tr>
<td>31</td>
<td>163 071</td>
<td>191.335</td>
<td>144</td>
</tr>
</tbody>
</table>

**Figure 4.6:** Number of LUT flip-flop pairs used as a function of the fractional word length

**Figure 4.7:** Number of block RAMs used as a function of the fractional word length
4.3 Execution time

As shown in Section 4.1, 20 fractional bits is enough to achieve close to full precision of the model. At that bit count, the maximum clock frequency of the model is 9 MHz.

The model used in this project model has $t = 680$ time steps, $n = 900$ states per time step and $i = 5400$ different inputs. The depth $d$ of the generated pipeline is 93 steps. Using Equations (2.4) and (2.5) which are repeated below, the total execution time of the optimisation problem can be computed.

The amount of clock cycles required if the pipeline is flushed after every state is:

$$T = tn \cdot (i + d)$$

Which means that $T = 680 \cdot 900 \cdot (5400 + 93) = 3 \, 361 \, 716 \, 000$ clock cycles are required. At 9 MHz, this would take $\frac{T}{9 \times 10^6} \approx 373$ seconds.

Similarly, the amount clock cycles required if the pipeline is flushed after every time step is:

$$T = t \cdot (ni + d)$$

which means that $T = 680(800 \cdot 5400 + 93) = 3 \, 304 \, 863 \, 240$ clock cycles are required. At 9 MHz would take roughly 367 seconds to compute. Flushing the pipeline after each time step would require some extra logic for keeping track of
which state each model execution corresponds to, but the amount of hardware required is most likely negligible compared to what is used for the model and the dynamic programming hardware itself.

For comparison, the original model runs in 251 seconds on a Intel Core i7-7500U CPU with no parallelisation.

4.4 Performance bottle necks

After initial synthesis, some bottle necks were identified and worked around in order to get an idea of the performance with some more work put into the project. Those bottle necks and workarounds are presented in this section. All synthesis here is done at 20 fractional bits.

4.4.1 Square root computation limits performance

According to the synthesis tool, the slowest part of the model is the computation of the square root which is a long combinatorial circuit. By pipelining this computation, the performance might improve. However, on its own, this turns out to not be the case, the minimum clock period is actually increased slightly, from 118 ns to 122 ns. The lack of an improvement is most likely caused by the square root not being an actual bottle neck, while the slight increase is possibly caused by the stochastic nature of HDL synthesis.

While the changes to clock frequency are mediocre, the changes to resource usage are much better. The amount of LUT flip-flop pairs is reduced from 101,849 to 66,267. However, when the square root computation is clocked, the lookup table uses block RAM instead of hardware which means that the total RAM usage changes from 95 to 100 block RAMs.

4.4.2 Divisions use a large amount of multiplexers

Division is currently implemented using the verilog division operator which generates a fully combinatorial circuit for the division. This is inefficient, both in terms of hardware usage and frequency, as each division in the model uses between 1000 and 3000 multiplexers.

Almost all divisions in the model are divisions by constants which can be replaced with multiplications by the inverse of the constant. In order to estimate the effect of that modification, the model was synthesised with all divisions replaced by multiplications.

At 20 fractional bits, without the modified square root hardware, this resulted in the amount of LUT flip-flop pairs used being reduced to 50,193 and the minimum clock period decreasing to 19.468 ns. This significant improvement indicates that the divisions are the main bottle neck of the design.

Additionally, when both of these optimisations are combined, the minimum frequency is reduced even further, to 10.346 ns and the number of LUT flip-flop-pairs is reduced to 8,832. This indicates, as suspected that the square root is suboptimal and that once the division bottle neck has been resolved, optimisations to the square root computation can further improve performance.
In this section, the results presented in the previous chapter will be discussed and analysed.

5.1 Execution time

It is clear from the results that the optimisation problem cannot be solved in real time using the current output from the HDL converter. However, there are some things which could improve performance. First, the frequency is bottlenecked by one or a few nodes which take a long time to compute and optimising those would improve the performance of the whole system. Some of those optimisations were discussed in Section 4.4. With more optimisation and support for more than one pipeline stage per node, it should be possible to run the design at the maximum frequency of the FPGA which is generally around 500 MHz.

Another thing which would improve the performance of the design is to evaluate multiple inputs in parallel. The amount of parallelism is currently limited by the fact that one state cost must be read in every model execution and RAM is limited to one or a few read ports. There are several methods to get around that, for example replication, banking or more advanced methods such as the one presented in [4].

5.1.1 Usefulness with further optimisations

Equations (2.4) and (2.5) can easily be modified to take parallelism into account. As a reminder, \( T \) represents the total amount of clock cycles required, \( t \) is the amount of time steps in the model, \( n \) is the amount of states per time step, and \( i \) is the amount of inputs to evaluate per state. Finally, \( d \) is the depth of the pipeline. For simplicity, only the hardware where the pipeline is flushed after each time step is considered which with \( p \) concurrent pipelines would require
\[ T = t \cdot \left( \frac{ni}{p} + d \right) \] (5.1)

clock cycles to compute.

Naturally, the FPGA area required would also increase by a factor \( p \).

At a given frequency \( f \), the time \( \tau \) required to complete \( T \) operations is given by

\[ \tau = \frac{T}{f} \]

This, along with equation (5.1) can be used to calculate the amount of concurrent pipelines \( p \) that would be required to run the optimisation in a specified amount of time at a given clock frequency as is shown in the following equation:

\[ \tau = \frac{T}{f} = \frac{t \cdot \left( \frac{ni}{p} + d \right)}{f} \iff \frac{f \tau}{t} = \left( \frac{ni}{p} + d \right) \iff \frac{f \tau - d}{ni} = \frac{1}{p} \iff \frac{f \tau - d}{ni} = p \]

Therefore, to run one full optimisation in one second at 100 MHz which is roughly the clock speed achieved when performing the optimisations mentioned in Section 4.4 would require

\[ p = \frac{900 \cdot 5400}{1.1 \times 10^8} - 93 = 33.07 \]

concurrent pipelines.

Naturally, fewer concurrent pipelines would be required if more optimisations are performed, for example, to run one full optimisation in one second at a clock frequency of 500 MHz, only

\[ p = \frac{900 \cdot 5400}{5 \times 10^8} - 93 = 6.6 \]

concurrent pipelines would be required.

### 5.1.2 Usefulness in its current state

Another thing to consider is whether or not the current implementation is useful in some cases, or if it has to have the above improvements before it can be useful. When comparing the execution time of the FPGA with the computation time on a single core CPU, the FPGA may seem completely pointless without optimisations. However, the FPGA still has an advantage: the computation time of the FPGA model is largely unaffected by the complexity of the model. The CPU
implementation would take twice as long to compute each iteration if the model execution time doubled, but the FPGA would barely be affected at all. Changes in model complexity would only affect the pipeline depth and resource usage, not the execution time.

This means that optimisation problems with fewer states but more complex models than the one used in this project would benefit more from FPGA hardware.

5.2 Model modifications

In this section, some modifications to the model which could be made in order to improve the model performance on an FPGA will be discussed.

5.2.1 Replace divisions with multiplication

As can be seen in Section 4.4.2, the resource usage and maximum frequency can be substantially improved by avoiding divisions. The divisions used in the model come in three categories: division by constants, division by values from lookup tables, and finally, division by calculated values.

Division by constants could easily be optimised by simply pre-calculating $\frac{1}{c}$ and replacing the division by a multiplication. This could even be done automatically by the conversion tool with some small modifications.

The second case, where the values to be divided by comes from the result of a function lookup, a similar method could be used. Instead of storing $f(x)$, the value of $\frac{1}{f(x)}$ could be stored and looked up directly. If $f(x)$ is only used for division, this would require no additional resources, otherwise it would use an additional block RAM to store the inverted values.

The final case, where the divisor is computed directly can not be converted into simple multiplication. This would most likely mean that those divisions still restrict the maximum frequency. However, more efficient division circuits are available, especially if pipelining can be used.

5.2.2 Avoid very large or small values

As has been shown, the model contains some values which are small and require a lot of fractional bits for storage. However, it also contains a lot of very large values which are likely stored at much higher precision than is required. If subexpressions where all values are large can be identified, it might be possible to temporarily scale them down while performing the calculation and then scaling them back up when they are combined with values which require more precision.

For example, consider the expression shown in Figure 5.1. Here, it might be possible to scale $V$ and $x$ to much smaller values, perform the calculation, and then scale the output back up before performing the square root operation. If the amount to scale by is a power of two, the scaling does not affect resource usage much, apart from adding an additional pipeline register.

It should be noted, that this will not be useful if per-node fractional word length
optimisation is implemented, as that would automatically scale values according to the required precision.

5.2.3 Use resources more effectively

While most hardware in an FPGA is completely reconfigurable, they also have dedicated hardware for things like RAM and multipliers. As these cannot be dynamically reconfigured to handle varying bit sizes, a computation that uses some of the device can use all of it without additional penalty. For example, the lookup tables that are used could have an even power of two number of entries in order to fit better in block RAMs. The size could also be rounded down if the data is close to fitting in a smaller RAM, at the cost of some output quality.

5.2.4 Use velocity as a state variable

The current model uses the kinetic energy of the vehicle as a state variable while a lot of the computations performed use the velocity. This requires the computation of the square root of a value which is currently a performance bottle neck. It might be possible to use the velocity as a state variable and compute the kinetic energy using a multiplier instead, which would get rid of one of the expensive square root computations. However, this would make the discretisation steps non-linear which introduces additional complexity, especially when interpolating between states to determine the cost of a previous state.
The primary result of this thesis is a C++ to HDL converter which can generate verilog code for an HEV model that can be used for dynamic programming. For the converter to work, a user has to replace the types used in the original C++ code with new types which represent an expression tree. This tree is then automatically converted to verilog. The type replacement is fairly straightforward and could most likely be done through text replacement as the standard arithmetic operators are overloaded. However, some more thought is required to replace some programming constructs which can not yet be fully converted.

In the current state, the generated code is too slow to run in real time, however, some optimisations to the converter are proposed that would significantly improve the performance. Additionally, the converter without optimisations might still be useful if expensive computations such as square root or division are not used, or if the model contains more calculations but fewer states than the model used in this thesis. This is because the model is fully pipelined, which makes the execution time largely independent of model complexity.

Finally, some possible changes to the model and optimisation problem are proposed which would make it more suitable for usage on an FPGA. Most of the proposed changes reduce the hardware usage by avoiding unnecessary expensive computations, for example replacing divisions by multiplications and changing the representation of one of the states from velocity squared to velocity. Another proposed modification is to change variable sizes to more efficiently use the available hardware in the FPGA. For example, if a function lookup table is stored in RAM, the size should be a power of two in order to not waste RAM resources.

Replacing divisions by multiplications would not affect the output quality at all, and could even be done automatically. Replacing the variables used as state
would also not affect performance if done correctly, however, it would make the spacing of the state non-linear which would require further modification. Finally, fully utilising the already used hardware would affect output quality. If the values are expanded to fully utilise already used hardware, it is likely that output quality would improve, while truncating value to fit less resources would most likely reduce output quality.
As always, things can be improved further, and this project is no exception. This final chapter will present some possible improvements to the HDL converter, as well as some model behaviour that could be researched further.

### 7.1 Avoiding divisions by constants

As was shown in Section 4.4.2, division consumes a substantial amount of hardware which can be optimised away by replacing division by multiplication. While this could be done manually, it would be tedious work which could be done automatically by the conversion tool.

### 7.2 Constant folding

A lot of programs are written to perform calculations where all operands are constant. The resulting values of those computations are also constant because they do not depend on any runtime values and can therefore be computed offline at compile time [11]. This process is called constant folding and results in more efficient execution. The synthesis tool seems to detect most constant calculations and optimise them away, however, doing it in the C++ converter can add additional division replacement possibilities.

Adding constant folding to the current system would require adding the ability to modify the expression graph once it has been generated, but should be fairly simple to implement when that has been done.
7.3 Avoiding duplicate nodes

In some cases, the same value is computed more than once in separate branches of the expression graph. By detecting such cases and unifying nodes where this happens, the resulting hardware can be made more efficient. Like constant folding, this requires the ability to change the expression graph after generation as well as some efficient way of comparing nodes.

7.4 Simulation on more input data

The results presented in this report are based on two separate but similar road profiles, and a single vehicle model. It would be useful to simulate the model on more road profiles to see if the required amount of fractional bits remains roughly constant, or changes depending on the road.

Additionally, it would be interesting to investigate if there is a large difference in performance between different vehicle models.

7.5 Fixed point model behaviour

The required amount of fractional bits for good model performance changes drastically between 19 and 20 bits for both road profiles. It might be interesting to investigate what causes this. Similarly, the velocity profile of the model at low bit counts often looks similar to the velocity at higher bit counts. Investigating why this is, and why it differs more in other regions might give hints for model changes that can reduce the amount of bits required.
Fractional word length optimisation

As mentioned in Section 3.7, the integer word length of all variables was optimised, but the fractional word length was not. However, some attempts at doing that were made and those attempts are documented in this chapter.

In order to select the size of each register in the design, a slightly modified version of the algorithm presented in by N. Doi et.al. was used [1].

Their method requires requires a maximum allowable error $E_{m}^{\text{max}}$ for each output $m$ as well as a function $E_{m}(L)$ which computes the error in output $m$ where $L$ is a vector representing the word length of each variable.

Using these, a set of bit lengths which uses the lowest amount of bits possible while ensuring that the output errors are not too large can be found by solving the following optimisation problem where $n$ is the total amount of nodes.

$$\begin{align*}
\text{minimise} & \quad \sum_{i=0}^{n} L[i] \\
\text{subject to} & \quad E_{m}(L) < E_{m}^{\text{max}} \forall m
\end{align*}$$

(A.1)

The next subsections present different sources of errors and ways of calculating their impact on the error in the outputs.

**A.1 Truncation errors**

In any node, the resulting value of the local operation can be truncated to a lower amount of bits. Each node has a maximum precision of its output which depends on the precision of its inputs. For example, an addition of two values with $n$
fractional bits has a precision of \(2^n\). Adding more than \(n\) bits to the output will not improve the precision, but bits can be removed in order to reduce the amount of bits used at the cost of precision.

In the worst case truncating a value to \(k\) bits when the output of the local computation has \(n\) bits adds an additional error, a truncation error, that is \(E_R = 2^{-k} - 2^{-n}\). Thus, the truncation error in node \(m\) with an operand \(k\) can be computed as \(E_R^m(L) = 2^{-L[m]} - 2^{-L[k]}\).

However, this calculation only works if \(L[m]\) is smaller or equal to \(L[k]\). Therefore, another constraint must be added to the optimisation which says that \(L[m] \leq L[k]\). This means that the final optimisation problem to be solved is

\[
\text{minimise } \sum_{i=0}^{n} L[i] \\
\text{subject to } E_m(L) < E_{m}^{\text{max}} \quad \forall m, \\
L[m] \leq L[k] \quad \forall (m, k) \text{ where } k > m \tag{A.2}
\]

where \(k > m\) means that the output of node \(k\) is truncated at node \(m\).

### A.2 Propagation errors

Additionally, errors propagate as operations are performed. A variable \(x\) can be estimated by \(\bar{x} + \Delta x\) where \(\bar{x}\) is an estimation of \(x\) and \(\Delta x\) is the error in that estimation.

When adding two variables, the result is

\[
y = \bar{x}_1 + \Delta x_1 + \bar{y}_2 + \Delta x_2 \Leftrightarrow E_p(y) = \Delta x_1 + \Delta x_2
\]

which means that the propagation error \(E_p^m\) in the result of an addition is the sum of the errors of the operands.

Similarly, the result of a multiplication is

\[
y = (\bar{x}_1 + \Delta x_1) \cdot (\bar{y}_2 + \Delta x_2) \Leftrightarrow E_p(y) = \bar{x}_1 \Delta x_2 + \bar{y}_2 \Delta x_1 + \Delta x_1 \Delta x_2
\]

which means that the propagation error of a multiplication is the sum of the errors multiplied by the operands. We can assume that the errors are small, which means that we can ignore the \(\Delta x_1 \Delta x_2\) term.

Similar arguments can be used to calculate the errors resulting from other operations which means that the propagation error of a node \(m\) is a function of the errors of all its operands.
A.3  Output error as a function of bit lengths

In order to optimise the amount of bits in each variable, the error in the output as a function of the bit length must be computed. This can be done recursively. The error in a node with no input operands is the truncation error of the input at the current bit length.

For other nodes, the error is the propagation error of the inputs plus possible truncation errors in the node:

\[ E_m(L) = E_m^R(L) + E_m^P(L) \]

A.4  Implementation

In order to perform this calculation, the node class was augmented with a function called `error_function`. It returns a closure which computes \( E_m(L) \), the maximum error in the node given a list of the bit length of each parent variable.

As explained in Section A.1, adding more bits than the amount of bits in the output of a node does not improve precision. Because of this, a second function called `smaller_word_constraints` was added to each node. This function returns a list of nodes which the node must be smaller or equal to.

Using these functions, an optimal value for the amount of bits for each variable should be computable by solving the optimisation problem (A.2). Initially, the `optimize.minimize` function in the python library `scipy` was used, and worked for small examples but did not converge when executed on the full model. Another attempt was made, this time using the `optimize.differential_evolution` function which may have produced useful results. However, there was not enough time to explore this further.

---

2. scipy.org


