Preprints of the 21st IFAC World Congress (Virtual)
Berlin, Germany, July 12-17, 2020

A Tool to Enable FPGA-Accelerated Dynamic Programming for Energy Management of Hybrid Electric Vehicles

Frans Skarman Oscar Gustafsson Daniel Jung Mattias Krysander

Department of Electrical Engineering, Linköping University, Linköping, Sweden (e-mail: {frans.skarman, oscar.gustafsson, daniel.jung, mattias.krysander}@liu.se)

Abstract: When optimising the vehicle trajectory and powertrain energy management of hybrid electric vehicles, it is important to include look-ahead information such as road conditions and other traffic. One method for doing so is dynamic programming, but the execution time of such an algorithm on a general purpose CPU is too slow for it to be useable in real time. Significant improvements in execution time can be achieved by utilising parallel computations, for example, using a Field-Programmable Gate Array (FPGA). A tool for automatically converting a vehicle model written in C++ into code that can executed on an FPGA which can be used for dynamic programming-based control is presented in this paper. A vehicle model with a mild-hybrid powertrain is used as a case study to evaluate the developed tool and the output quality and execution time of the resulting hardware.

Keywords: Hybrid vehicles, Dynamic programming, Energy management systems, Computer-aided circuit design, Integrated circuits

1. INTRODUCTION

By adapting the driving pattern and powertrain energy management of hybrid electric vehicles to the road ahead and traffic conditions, the fuel consumption and emissions of the vehicle can be reduced. Connected vehicles and infrastructure (V2X), GPS, and route data, give relevant information for calculating the optimal split between the use of the electric motor and combustion engine, for example using Dynamic Programming (DP) (Pérez et al., 2006). However, the runtime of a DP algorithm grows exponentially with the number of inputs and state variables which makes its use for real time calculations in a Hybrid Electric Vehicle (HEV) difficult on a general purpose CPU (Sciarretta and Guzzella, 2007).

Due to the computational complexity, DP is often only applied for off-line analysis and to generate benchmark performance results, see e.g., Wang and Lukic (2012). There are real-time implementations of DP for conventional powertrains, see for example Hellström et al. (2009), where the model complexity is less than, for example, hybrid electric powertrains. Different approaches have been proposed to reduce computational complexity, for example using efficient search strategies (Hellström et al., 2010). In Lock and McKelvey (2017), an iterative DP approach is applied where a finer and finer search grid is used around the previous solution to reduce the overall complexity. Also, connected vehicles and cloud-computing have been proposed to have access to additional computational power (Ozatay et al., 2014). Another option which is explored here is to exploit the high degree of parallelism present in DP algorithms by running them on a Field Programmable Gate Array (FPGA). While the primary focus here is to run the optimisations locally in a vehicle, an FPGA implementation would also be beneficial in a cloud computing scenario as FPGAs have a much lower power consumption than CPUs and GPUs.

FPGAs are configurable integrated circuits allowing them to make computations without the overhead of fetching and executing instructions. They are also able to efficiently exploit parallelism which makes them suitable for this application. A potential downside of FPGAs is that programming them requires a different skill set than general purpose processors. In order to mitigate that, a tool is proposed in this paper, Cinnabar[1], which converts a vehicle model written in C++ into code in a Hardware Description Language (HDL) that can be executed on FPGA hardware. The process requires some manual steps, but no knowledge of FPGA programming.

A case study of a mild-hybrid electric vehicle is used in this paper to evaluate the performance and output quality of the DP algorithm when the vehicle model is run on an FPGA.

2. PROBLEM STATEMENT

In order to run the DP algorithm for vehicle speed optimisation and powertrain energy management on an FPGA, a model of the vehicle that can be executed on an FPGA must be created. However, implementing algorithms on an FPGA requires different skills compared to implementation on a general purpose processor. Therefore, a tool that

[1] *https://gitlab.com/cinnabar/cinnabar*

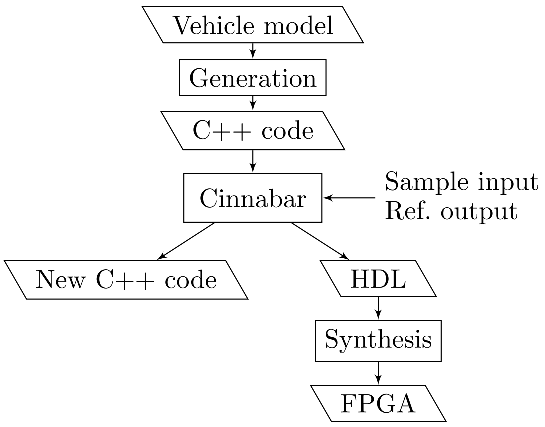

Fig. 1. Overview of the model conversion process.

can automatically convert models used for simulation and control, into code that can be run on an FPGA would help save development time. A tool, Cinnabar, which automates most of this process is presented in this paper.

Figure 1 shows an overview of the process required to use the tool. Since vehicle models often already exist as code in high level languages such as Matlab or C++, the goal of Cinnabar is to convert high level code, in this case C++, into code that can be executed on an FPGA. Most of the process is fully automatic, but for now, some manual work is required.

The process starts with expressing the vehicle model in C++ code using special tool defined classes. This code is passed to the tool along with sample model inputs and outputs for use in verification and optimisation. Using the sample inputs and the C++ code, the tool outputs HDL code which can be synthesized and executed on an FPGA. It also outputs new C++ code which emulates the HDL code. This new code allows further simulation, or verification of the model.

The output quality and amount of FPGA resources used by the model depends in a large part on the data types used for the computations. Therefore, an investigation is conducted using a case study to investigate how many bits are needed for the computations without significantly reducing the output quality.

3. POWERTRAIN ENERGY MANAGEMENT USING DYNAMIC PROGRAMMING

The problem of optimising the fuel efficiency of an HEV for a given route is to find a power split strategy, gear shift sequence, and velocity profile that minimizes fuel consumption while fulfilling given constraints.

Optimisation constraints include speed limits, allowed variations in battery State of Charge (SOC), and travel time. Since route constraints, such as speed limits and locations of stop signs, depend on travelled distance, the optimisation problem is formulated with respect to travelled distance instead of time (Hellström et al., 2009). The constraints as a function of travelled distance are illustrated in Fig. 2 where vehicle velocity is used as the only state. With the car at a specific velocity and distance from the start, each of the control inputs will put the car in another state. The path to get to the current state is

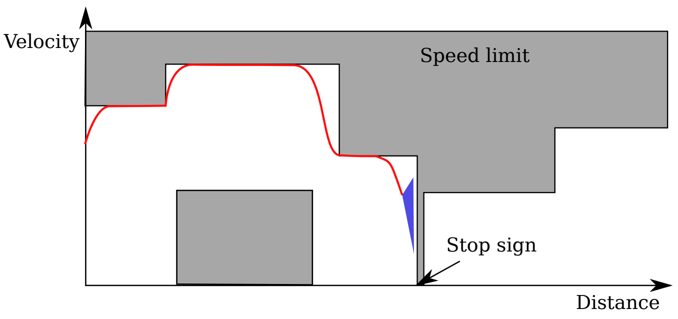

Fig. 2. State space for a vehicle with velocity as the only state variable.

represented by the red line, and the effect of the possible control inputs is represented by the blue triangle. The infeasible states are marked with grey colour and, in this case, correspond to speed limits and stop signs.

3.1 Dynamic Programming

DP is an efficient exhaustive search method that solves optimisation problems by recursively solving a set of simpler sub-problems (Bellman et al., 1954). In this case, if the cost of getting from a state x_2 to the goal is known, and an input u takes the vehicle from state x_1 to x_2, then the cost of x_1 using input u is the cost of that input plus the cost of x_2. This cost c_{state} can be computed recursively from the goal and backwards as

$$c_{\text{state}}(x_i) = \min_{u \in \text{inputs}} \left(c_{\text{input}}(u) + c_{\text{state}}(s(x_i, u)) \right) \quad (1)$$

where c_{input} computes the cost of a specified input, c_{state} is the cost of the specified state, also referred to as cost-to-go, and $s(x, u)$ computes the resulting state when applying input u in state x.

Finding an optimal solution requires evaluating every input in every state, which requires a lot of calculations. However, if there is no way to go between two states, those two states can be evaluated in parallel. This is the case here since a vehicle can not change velocity or state of charge without also moving forward in time. This inherent parallelism is the primary motivation for exploring the usage of FPGAs for this optimisation.

4. FPGA

A Field Programmable Gate Array (FPGA) is a chip consisting of a large number of programmable logic blocks which are connected by reconfigurable interconnect. By changing the behaviour of the logic blocks and the connections between them, the FPGA can be configured to perform arbitrary computations.

FPGAs have a number of advantages and disadvantages over conventional computers. One such advantage is that they can be programmed to perform a specific computation without the overhead associated with a general purpose processor which has to fetch and parse instructions. They are also able to perform many calculations in parallel as each logic block acts independently.

FPGAs can exploit parallelism in a computation by using pipelining. Each step of the computation to be performed is given dedicated hardware where the result of the previous step is fed into the next through registers. For example,

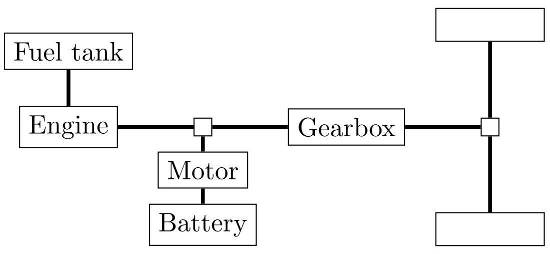

Fig. 3. Schematic view of the vehicle being modelled.

if $(a + b) + c$ is to be computed for several values of a, b, and c, the value of $a_1 + b_1$ can be computed in one adder while $(a_2 + b_2) + c_2$ is being computed in another.

This works until there is a dependency between values, for example if a is the result of the previous sum in the above example. At that point, the pipeline has to be *stalled* until the previous computation is finished.

4.1 Related Work

There has been some previous research into using FPGAs for DP. For example, Hu and Georgiou (2013) presented an FPGA adaptation of a genome sequencing algorithm. The algorithm is different to the one used for vehicle optimisation which means that most of their findings are hard to apply here.

Settle (2013) also performed genome sequencing using dynamic programming on an FPGA. Unlike the previously mentioned paper, they used OpenCL for programming, which removes the need for knowledge of FPGA programming. However, it instead requires knowledge of OpenCL.

There are also several previous studies of running dynamic programming on other parallel hardware. For example, Cruz et al. (2014) presents a method for efficiently running dynamic programming on a Very Large Instruction Word (VLIW) processor. Miyazaki and Matsumae (2018) presents a pipelined DP implementation for Graphics Processing Units (GPUs). However, since the papers investigate different kinds of hardware for other problems, their findings are not directly applicable to this work.

Using a high level language to write code for FPGAs is not a new concept, and there is a wide variety of tools available for the task. For an overview, see (Nane et al., 2016) in which the authors conducted a survey of some of the available tools. Unlike most other High Level Synthesis (HLS) tools, Cinnabar automatically selects the data types used for the computations. Additionally, it is focused on models for use with dynamic programming, rather than arbitrary code.

5. VEHICLE MODEL

This section summarises the vehicle model used in the case study to illustrate some of the component models that are used for look-ahead powertrain energy management applications, e.g. in Jung et al. (2018). The objective is to highlight the kind of computations that the tool must support for executing the model. For a more detailed description of the powertrain and component models, the reader is referred to, e.g., Jung et al. (2018).

The case study is a mild parallel HEV as illustrated in Fig. 3, but the methods can be applied to other kinds of powertrain architectures as well. The model has three inputs: gear, electric motor torque, and combustion engine torque which is combined with braking force. It also has two states: kinetic energy, and battery SOC. Kinetic energy is used as state instead of velocity as recommended in Hellström et al. (2010).

The vehicle model computes the average velocity and fuel consumption for the given inputs for a given interval. These are used to compute the next state, and cost of the inputs which are then used in (1). Since the states are parametrised by distance, the cost of a state is computed as

$$c_{\text{state}}(x) = \frac{\Delta d}{v_{\text{avg}}}(\dot{m}_{\text{fuel}} + \gamma), \tag{2}$$

where Δd is the distance between states, v_{avg} is the average velocity during the interval, and $\dot{m}_{\text{fuel}}$ is the amount of fuel consumed in the interval. Finally, γ is a tuning parameter which determines how to prioritise arrival time over fuel consumption (Hellström et al., 2009).

The longitudinal vehicle dynamics at distance step i are computed from kinetic energy as

$$v_{i+1}^2 = v_i^2 + \frac{2}{m}\Delta d\left(F_{\text{trac},i} - F_{\text{a},i} - F_{\text{r},i} - F_{\text{m},i}\right),$$

where m is vehicle mass, v is speed, F_{trac} is the traction force generated by the powertrain, F_{a} is aerodynamic friction, F_{r} is rolling friction, and F_{m} is rolling resistance (Sciarretta and Guzzella, 2007).

The transmission is modelled using the relations $w_{\text{pt}} = \alpha w_{\text{wh}}$, and

$$T_{\text{wh}} = \begin{cases} \alpha\eta T_{\text{pt}} & \text{if } T_{\text{pt}} \geq 0 \\ \dfrac{\alpha}{\eta} T_{\text{pt}} & \text{otherwise,} \end{cases} \tag{3}$$

where T_{pt} and w_{pt} are powertrain torque and speed, T_{wh} and w_{wh} are wheel torque and speed, respectively, α is the selected gear ratio between motor and wheel rotation, and η is the efficiency of the gearbox.

The electric motor efficiency and combustion engine fuel consumption are modelled with static efficiency maps as functions of motor/engine speed and torque.

The current consumed by the electric motor is computed by

$$I_{\text{em}} = \frac{V_{\text{oc}}(\text{SOC}) - \sqrt{V_{\text{oc}}(\text{SOC})^2 - 4R_0P_{\text{em}}}}{2R_0(\text{SOC})}, \tag{4}$$

where V_{oc} and R_0 are the open circuit voltage and internal resistance of the battery, respectively, while P_{em} is the desired power from the electric motor.

The battery SOC during interval i is updated as $SOC_{i+1} = SOC_i - \Delta dI_{\text{em}}/(v_{\text{avg}}C)$ where C is the battery cell capacity.

All of these equations contain standard arithmetic operations on both constants and variables which therefore has to be supported by the tool. In addition, (4) requires computing both the square root, and square of values. The values $V_{\text{oc}}(\text{SOC})$ and $R_0(\text{SOC})$ are functions of the battery SOC. However, instead of being computed directly their values determined by a lookup table with interpolation between values. The tool supports this for both 1D and 2D functions. Finally, to compute (3), support for conditional values and boolean operators is required.

6. VEHICLE MODEL TO FPGA CONVERSION

Cinnabar requires a version of the vehicle model written in C++ using some tool defined classes. The following code snippet shows a comparison between a normal C++ function and a function using the tool defined classes.

```
// Original function using normal C++ data types
float example(float x, float y) {
    float product = x * y;
    if (x > y) {
        return product + x;
    }
    return product - y;
}
// Cinnabar version of the same function
Node example(Input x, Input y) {
    Node product = x * y;
    return _if((x > y),
        product + x,
        product - y
    )
}
```

The first thing to notice in this example is that the original code and the Cinnabar version are structurally similar. The basic operands are exactly the same, apart from working with different data types. Some operations however, like the if-statement can not be directly executed on an FPGA as they can not easily conditionally execute code. Instead, both branches are computed, and the condition decides which value to return. This operation is represented by the `_if`-function.

In addition to the C++ code, the tool must be provided with sample inputs and outputs for the model. From that, the tool determines values for the Fractional Word Lengths (FWLs) which ensures a low enough error while keeping resource usage low.

The sample input provided should be representative of the actual input that the model will receive when in use. If this is not the case, the resulting code might produce worse results than desired in some situations that were not covered by the sample input.

6.1 Cinnabar Internals

Figure 4 shows an overview of the process that Cinnabar uses to convert C++ code into HDL code. As explained in the previous section, the tool is fed with C++ code using tool defined classes. While the classes behave like normal primitive data types to a user, their internal representation is quite different. Rather than representing values, the types represent computations. For example, the result of an addition is not the value of the sum of the operands, it is the computation of that sum.

These classes together build up an expression graph, a structure which describes all the operations required to perform a computation. This basic expression graph is most likely not optimal, for example, there may be unnecessary computations, or computations that can be made more efficient in hardware. To mitigate this, the graph is transformed using various optimisations to yield a new and improved expression graph.

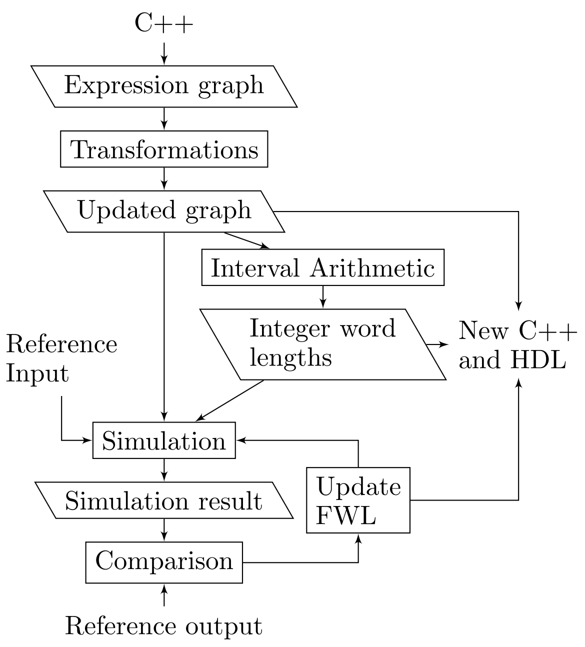

Fig. 4. Overview of the conversion process used by Cinnabar.

One important step in designing FPGA hardware is to determine what data types to use for storage and computations of values. Typically, fixed point values are preferred instead of floating point as the resulting hardware for performing computation on them is simpler and uses fewer resources.

However, to use fixed point values the tool must determine how many bits to use. The required integer word length for each register can be determined from the size of the values to be stored there. If the size of the inputs is known, the size of all other values can be computed recursively using interval arithmetic. For example, the bounds on the size of the result of an addition is the sum of the bounds on the operands.

Determining the amount of fractional bits required is more challenging as errors introduced through quantisation propagate through to the result. The tool simulates the model on the provided test data using different fractional word lengths. Once the simulation is done, the user can select a FWL setting that provides a good trade-off between output quality and resource usage. Currently, the same amount of fractional bits is used for all values.

Each operation in the graph is mapped to a unique component in the resulting hardware. Registers are inserted between components in order to ensure that inputs arrive at the correct time.

6.2 Executing the Vehicle Model

Once hardware for the vehicle model has been generated, it is connected to hardware which performs the dynamic programming described in Section 3.1. An overview of the hardware for doing so is shown in Fig. 5.

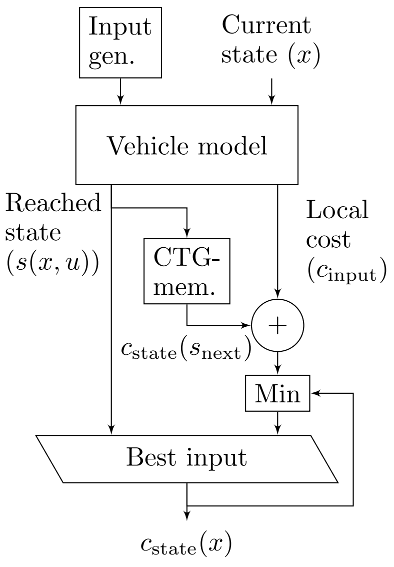

Fig. 5. Schematic view of the dynamic programming computation.

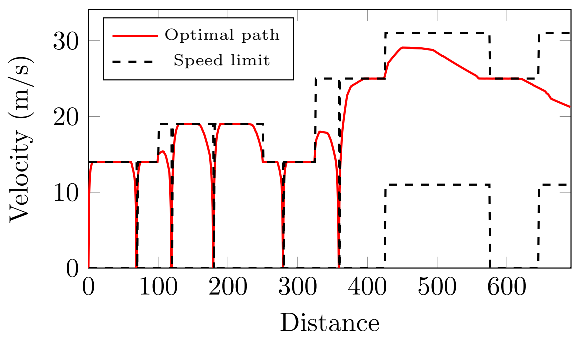

Fig. 6. One scenario used for performance evaluation.

In each state, the cost and resulting state of every input must be evaluated, which means that the vehicle model is executed once for each input in each state. The resulting state is used to look up the cost of the reached state in the cost-to-go memory (CTG-mem.) and the resulting cost is added to the cost of the input. Once all inputs are evaluated, the minimum value of the total cost is stored as the cost of the current state.

7. PERFORMANCE EVALUATION

In order to evaluate the performance of the tool and the resulting code, data from two test scenarios was used. One of the profiles is the same as the one used in Olin et al. (2019), while the other was created for this project and is shown in Fig 6. The routes in each scenario are discretised into 680 distance steps where the road profile consists of a mix of urban and highway driving.

The model was originally written in C++ using standard data types, and was then converted to the Cinnabar data types. The original model output was used to generate the reference output.

7.1 Output Quality

Figure 7 shows the resulting cost of journeys through the two different driving scenarios as a function of the amount

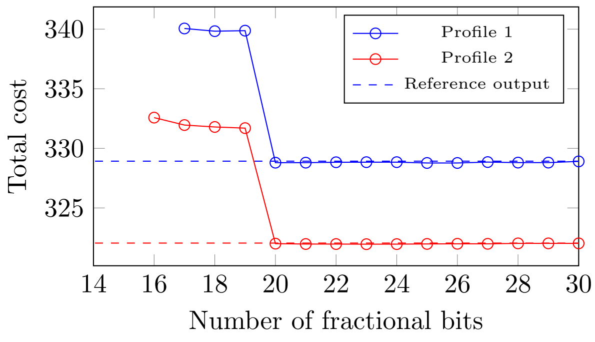

Fig. 7. Optimisation output when the model is run with different amounts of fractional bits.

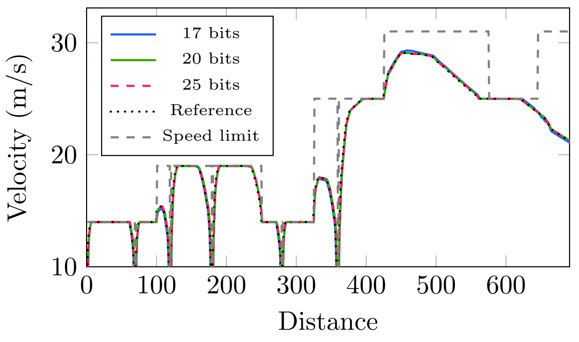

Fig. 8. Velocity profile at various fractional bit counts. As the difference is subtle, only a small section of the full path is shown.

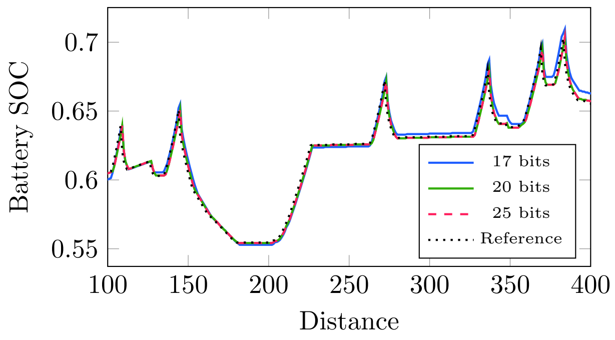

Fig. 9. State of charge profile over the same distance as shown in Fig. 8 at various fractional bit counts.

of fractional bits used for the computation. The dashed line represents the output of the original model.

The two graphs show a similar shape, with errors decreasing slightly as the fractional bit count approaches 20, at which point the cost of the fixed point version almost perfectly matches the reference. The simulation was run for 10 to 30 fractional bits, but no feasible solution was found by the model below 16 and 17 bits respectively.

Figures 8 and 9 show the velocity and battery state of charge profiles of the resulting paths for different word lengths. The profiles at 20 and 25 fractional bits are very close to the reference profiles. The velocity profile is also fairly close, even at 17 bits, while the state of charge profile deviates more noticeably at 17 bits. This is consistent with the conclusion that 20 fractional bits is enough for outputs which are very close to the original model.

7.2 Runtime Performance

In the model used throughout this work, all states in a single distance step can be computed in parallel. The generated FPGA hardware is pipelined which means that one model execution can start and finish every clock cycle apart from when flushing the pipeline between distance steps. This means that the amount of clock cycles required for an execution of the DP algorithm is $t = D \cdot (SN + P)$, where D is the amount of distance steps, S is the amount of states in a single distance step, N is the amount of inputs to test, and P is the pipeline depth.

The sample model used has 680 time steps which each contain 900 combinations of states, and in each state, there are 5400 inputs to test. The depth of the pipeline generated by Cinnabar for the sample model is 94 stages, which results in

$$t = 680 \cdot (900 \cdot 5400 + 94) = 3\ 304\ 863\ 920 \text{ clock cycles.}$$

At 300 MHz, a typical maximum frequency for many FPGAs, one full model evaluation would then take $\frac{T}{300 \cdot 10^6} \approx 11$ s. This is likely too slow for real-time usage, however, it is possible to further exploit the parallelism by adding concurrent pipelines. For example, with 11 parallel pipelines, each execution of the DP algorithm would only take 1 s, though it also requires 11 times more hardware.

Even without the concurrent pipelines, the code is much faster than the corresponding CPU model which runs in roughly 250 s on an Intel Core i7–7500U without exploiting any parallelism.

It is worth noting that the runtime of the FPGA version is independent of the complexity of the model, one model execution can always be done almost every clock cycle (apart from when waiting for the pipeline to be flushed). This allows addition of additional information to the model, such as road conditions, traffic, and plans of other connected vehicles without affecting runtime. It also means that making the model less complex will not speed up execution, the only ways to do that are to increase the clock frequency, further increase parallelism, or reducing the amount of states.

8. CONCLUSIONS AND FUTURE WORK

DP is a useful tool for analysing optimal energy management strategies for HEVs. However, the application of DP for real-time control is limited by its computational complexity. The computation time of DP can be significantly reduced by utilising parallel computations and one method for doing so is to execute it on an FPGA. The developed tool, Cinnabar, allows creation of FPGA code from a vehicle model with little manual work, and no knowledge of FPGA programming. The analysis shows the potential of the resulting hardware for real time vehicle applications.

For future work, further improvements of the tool are considered to reduce the amount of FPGA resources used. For example: selecting the amount of fractional bits used for each value independently, rather than using the same FWL for every value. It is also worth investigating making changes to the model which would benefit the FPGA implementation. For example, reducing the amount of states used, changing the lookup tables used for functions, or replacing lookup tables with normal function evaluations. Another improvement to be considered is to generate the Cinnabar C++ code from other representations, for example a model designed in Simulink.

REFERENCES

Bellman, R. et al. (1954). The theory of dynamic programming. *Bulletin of the American Mathematical Society*, 60(6), 503–515.

Cruz, M., Tomas, P., and Roma, N. (2014). Low-power vectorial VLIW architecture for maximum parallelism exploitation of dynamic programming algorithms. In *Proc. Int. Conf. on High Performance Computing & Simulation (HPCS)*, 88–95.

Hellström, E., Åslund, J., and Nielsen, L. (2010). Design of an efficient algorithm for fuel-optimal look-ahead control. *Control Eng. Practice*, 18(11), 1318–1327.

Hellström, E., Ivarsson, M., Åslund, J., and Nielsen, L. (2009). Look-ahead control for heavy trucks to minimize trip time and fuel consumption. *Control Eng. Practice*, 17(2), 245–254.

Hu, Y. and Georgiou, P. (2013). A study of the partitioned dynamic programming algorithm for genome comparison in FPGA. In *Proc. IEEE Int. Symp. Circuits and Systems*, 1897–1900.

Jung, D., Ahmed, Q., Zhang, X., and Rizzoni, G. (2018). Mission-based design space exploration for powertrain electrification of series plugin hybrid electric delivery truck. In *SAE Technical Paper Series*.

Lock, J. and McKelvey, T. (2017). A computationally fast iterative dynamic programming method for optimal control of loosely coupled dynamical systems with different time scales. *IFAC-PapersOnLine*, 50(1), 5953–5960.

Miyazaki, M. and Matsumae, S. (2018). A pipeline implementation for dynamic programming on GPU. In *Proc. Int. Symp. Computing and Networking Workshops*, 305–309.

Nane, R. et al. (2016). A survey and evaluation of FPGA high-level synthesis tools. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 35(10), 1591–1604.

Olin, P., Aggoune, K., Tang, L., Confer, K., Kirwan, J., Deshpande, S.R., Gupta, S., Tulpule, P., Canova, M., and Rizzoni, G. (2019). Reducing fuel consumption by using information from connected and automated vehicle modules to optimize propulsion system control. In *SAE Technical Paper Series*.

Ozatay, E. et al. (2014). Cloud-based velocity profile optimization for everyday driving: A dynamic-programming-based solution. *IEEE Trans. Intell. Transp. Syst.*, 15(6), 2491–2505.

Pérez, L.V., Bossio, G.R., Moitre, D., and García, G.O. (2006). Optimization of power management in an hybrid electric vehicle using dynamic programming. *Math. Comput. Simulation*, 73(1-4), 244–254.

Sciarretta, A. and Guzzella, L. (2007). Control of hybrid electric vehicles. *IEEE Control Syst. Mag.*, 27(2), 60–70.

Settle, S.O. (2013). High-performance dynamic programming on FPGAs with OpenCL. In *Proc. IEEE High Perform. Extreme Comput. Conf.(HPEC)*, 1–6.

Wang, R. and Lukic, S.M. (2012). Dynamic programming technique in hybrid electric vehicle optimization. In *Proc. IEEE Int. Electric Vehicle Conf.*, 1–8.