[HKK+22] M. Hoppe, J. C. Kirchhof, E. Kusmenko, C. Y. Lee, B. Rumpe:
Agent-Based Autonomous Vehicle Simulation with Hardware Emulation in the Loop.
In: 2022 IEEE Intelligent Vehicles Symposium (IV'22), pp. 16-21, IEEE, Jun. 2022.
https://se-rwth.github.io/publications/

Agent-Based Autonomous Vehicle Simulation with Hardware Emulation in the Loop

Mattis Hoppe, Jörg Christian Kirchhof, Evgeny Kusmenko, Chan Yong Lee, Bernhard Rumpe
Chair of Software Engineering, RWTH Aachen University

***Abstract*— Agent-based simulation is an important testing tool for the development of autonomous vehicle software. Simulators enable engineers to test autonomous driving behavior in virtual environments, which is cheaper, faster, and safer than using a physical vehicle. An important aspect of autonomous driving software is its real-time capability, i.e. its ability to react to unforeseen events and new sensor inputs within a very short amount of time to prevent accidents. In this paper, we present a modular agent-based simulator architecture, which not only simulates the physical behavior of the vehicle, controlled by the software under test, but also its electrical/electronic (E/E) network. In particular, each ECU is simulated using a hardware emulator, which enables us to test the software as if it is run on the actual target hardware. Furthermore, the hardware emulator estimates the execution delays for the software under test, which enables more realistic approximations of the real behavior. In an evaluation example we analyze empirically how well the timing estimates reflect the reality. We show that modeling the memory hierarchy and instruction decoding has a crucial effect on the precision of this estimation.**

I. Introduction

Software errors in autonomous vehicles can lead to severe consequences. In addition to the financial consequences of an accident, in extreme cases such errors can lead to the loss of human life[1]. Accordingly, it is essential to extensively test the software of autonomous driving vehicles in advance of their use on the road. Simulations offer a useful environment for such testing, as simulations not only provide reproducible results, but are also typically less expensive than testing with real vehicles.

For testing purposes, simulations contain a real system under test that is connected to a simulated environment. The real system does not have to be a complete autonomous vehicle, but can also be a software, a model, or a piece of hardware. Simulators are often categorized using the type of real system that they provide an environment for. These categories are called *X-in-the-loop*, where *X* represents the real system, *e.g.,* software-in-the-loop, model-in-the-loop, or hardware-in-the-loop [1]. For example, in a hardware-in-the-loop simulation could use a real vehicle controller that interacts with a simulation that provides, *e.g.,* sensor inputs. Since autonomous vehicles must share the road with other vehicles, simulators for autonomous vehicles must also be able to represent the interaction between multiple entities making decisions independently. Simulations that can represent such independent actors are called *agent-based*.

While simulations are useful for testing software, they naturally abstract from the real world, since simulations only recreate the real world. Such abstractions can falsify the results of the simulation. Therefore, it is necessary to ensure that the results of a simulation are as close as possible to the results of a real world test. An important factor in this abstraction is the time consumed by hardware to execute a piece of functionality. For example, if an autopilot software can reliably calculate steering commands but takes two minutes for each calculation, it might not be useful in the real world. For such calculations, simulations can use hardware emulation to calculate the time consumption for executing a function on a specified hardware [2]. Albeit modelling some parts of the hardware, their approach ignores some important factors that contribute to the time consumption of executing instructions. In this paper we extend their approach by emulations for a memory cache hierarchy and instruction latency respecting the operands of an instruction. Thereby, we can improve the accuracy of the emulation.

II. Related Work

There are many approaches to emulating particular parts of automotive systems, *e.g.,* batteries [3] or power-line communication [4]. For autonomous driving, it is crucial that the processor can calculate driving commands fast enough to safely drive the vehicle to its destination.

The estimation of whether a processing is fast enough to execute a task in a given time is often based on the so-called worst case execution time (WCET). The WCET is often used in static or dynamic program analysis to estimate if the software can be executed fast enough. aiT [5] builds a control flow graph from the software to be analyzed. After building the control flow graph, aiT analyzes the used memory ranges of instructions that access memory and uses that information for an analysis of which accesses are cache hits/misses and how the program behaves based on its processor pipeline. This analysis is then used to calculate the WCET. If the WCET is based on measurements, it is important to also consider the probability with which the WCET might be exceeded [6]. For a simulation of multiple autonomously acting vehicles, such an analysis is likely infeasible as the number of paths a program can take through the control flow graph is very high. Compared to such static analyses, our approach is better suited for the testing use case as we only analyze concrete executions of the system under test.

This work was supported by the grant SPP1835 of the German Research Foundation (DFG).

[1]Neal E. Boudette: "'It Happened So Fast': Inside a Fatal Tesla Autopilot Accident", New York Times, 17.08.2021. [Online]. Available: `https://www.nytimes.com/2021/08/17/business/tesla-autopilot-accident.html` Last accessed: 01.02.2022

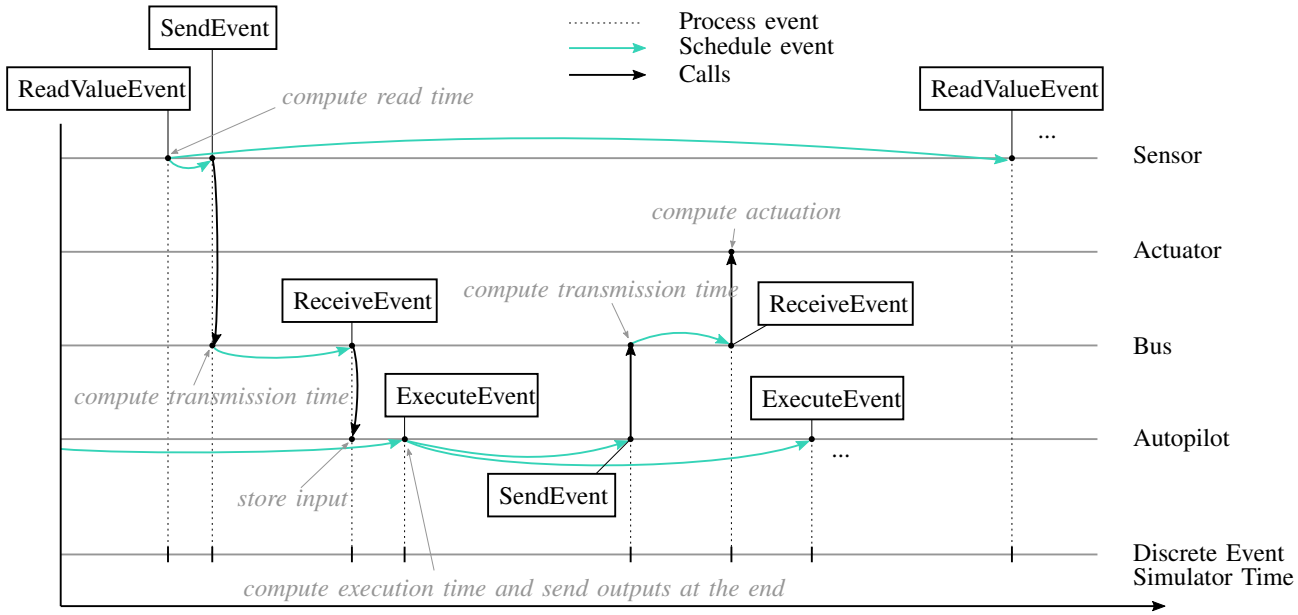

Fig. 1. MontiSim uses a discrete event framework to model communication between dynamic constituents of the vehicle.

Related work examined the combination of simulation and hardware emulation in various domains. For example, [7] provides a simulation of power grids that incorporates a library of emulated field-programmable gate arrays (FPGAs). In [8], the network simulator ns-3 is extended with a measurement-based emulator of wireless open access research platform (WARP) boards. In [9], a rotor blade is emulated as part of a wind turbine simulator. While hardware emulation and simulation have been combined in various other simulators, emulating the processing hardware of the vehicle to increase the validity of driving simulations did not receive enough attention. With a similar goal as our approach, [10] presents a hardware-in-the-loop simulator for autonomous driving. By incorporating real hardware in the simulation, their approach does not have to verify that the execution matches the execution time on a real device. In [11], a *vehicle-in-the-loop* simulator is presented that incorporates a test vehicle connected to a simulated environment. However, using real hardware limits the number of tested vehicles in both of these approaches to the available hardware. Using an emulation, our approach is only limited by the amount of available processing resources. If the simulation is executed as a service [12] in cloud systems, this effectively means the number of tested vehicles is only limited by the cost limit. The cost for this scalability is, however, the need for a thorough validation of the emulation.

III. MontiSim Toolchain

MontiSim is an agent-based intelligent transportation system (ITS) simulator for the testing of autonomous driving software [12]–[14]. This means, each vehicle in the simulation is an entity with its own perception of the world (created by its respective sensors) and its own intelligence controlling the actuators. The driving intelligence, i.e. the software under test, is an exchangeable component and can be plugged in before the simulation. Such a component has to fulfill a particular interface, but can otherwise be any piece of software mapping the sensor signals of the vehicle to actuator commands. The interface requires an `init()` function which is called at start up, an `execute()` function which is called each time the simulated hardware is ready to execute code, setters to provide sensor values to the autopilot and getters to read the resulting actuator commands. The component modeling framework EmbeddedMontiArc [15] [16] supports the automated generation of MontiSim interface code and provides means for behavior modeling, including deep and reinforcement learning [17] [18] [19].

To make the data processing and data transmissions, *e.g.,* between sensors and the autopilot, more realistic in terms of delays, sensors, transmission channels, and electronic control units (ECUs) are modeled as components and attached to the vehicle's E/E system. The E/E system and components attached to it follow a discrete event simulation approach. Whenever MontiSim advances its time, a time event is created and dispatched to all listeners. Sensors, actuators, busses, and ECUs are informed and check whether there are tasks for them to complete. For instance, a sensor readout might need to be sent to the autopilot ECU. Instead of accomplishing the transmission directly, the transmission component in charge, *e.g.,* a bus, estimates how long the transmission should last and schedules delivery for the corresponding point in time. Once this point in time is reached, the data is copied to its recipient. The overall idea of event-driven message passing is depicted in Fig. 1. This way, complex data transmission is taken into account and influences the execution of the autopilot accordingly. While the discrete event simulation is going on, the host simulator is blocked and will not advance time further until all components have finished their event scheduling and/or data processing. This

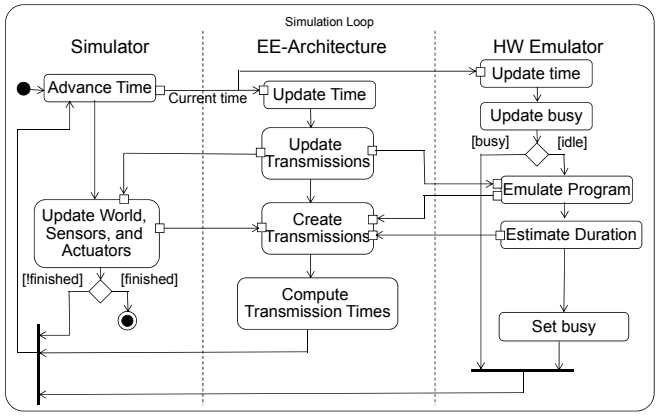

Fig. 2. Activity diagram: the core discrete-time simulator (left lane) simulates the physical world. Advancing time results in an event which the E/E architecture (middle lane) and the hardware emulator (right lane) need to process for each vehicle before the core simulator can proceed.

ensures that the simulation results are by no means influenced by the executing host hardware or the operating system and are completely reproducible.

One of the most interesting components in the E/E network of a MontiSim vehicle and also the main subject of this work is the hardware emulator. The hardware emulator component represents an ECU and runs the software under test. In case of multiple ECUs, multiple hardware emulator instances can be attached to the E/E network.

The communication pattern between the host simulator, the E/E architecture, and the emulated ECUs resulting from the event-based architecture described above is sketched in the activity diagram in Fig. 2. As all discrete-event sub-simulators, the E/E architecture and the hardware emulator get notified when time is advanced, i.e. when a new time step needs to be simulated. The E/E architecture schedules transmissions of newly seen data, *e.g.,* from sensors to the autopilot ECU. The data for these transmissions is held back until the simulation reaches the scheduled point in time for delivery. Then it passes the data to the intended recipient. Similarly, the hardware emulator emulates the autopilot program, estimates the execution duration (cf. Sec. IV for more details), and remains idle, waiting for the estimated execution time to pass (in simulation time) until it writes its result back to the E/E architecture to transmit it to the target device (*e.g.,* an actuator). While the hardware emulator waits for the scheduled time step for result delivery to arrive, its state is set to busy and it does not start new computations. Once the E/E architecture and the hardware emulator have finished their time step, control is given back to the host simulator, which updates its physics and initiates the next time step.

IV. Hardware Emulation

We base our hardware emulator design on the Unicorn-based concept presented in [2] and hence omit a detailed presentation thereof.

However, the framework presented in [2] neglects two major factors influencing the computation time: the presence

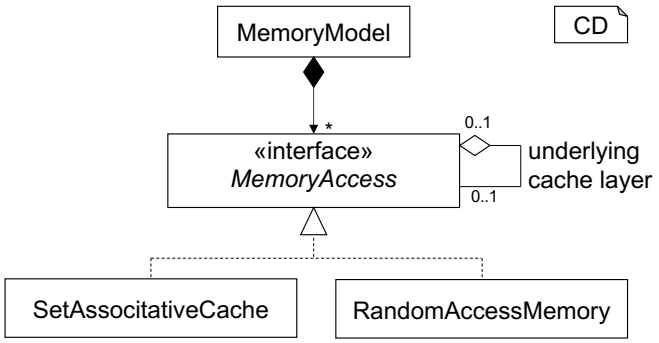

Fig. 3. High-level overview our cache hierarchy model. Each cache layer can have an underlying cache layer that takes over in case of a cache miss.

of a cache hierarchy and the decoding of instructions. This section presents our approach to modeling these two influences in the hardware emulator.

A. Modeling Cache Hierarchy

The faster the memory, the more expensive and smaller it is. Therefore, computers rely on architectures in which several layers of different fast memories work together. The faster the memory, the less of it is used. Because of the limited size, not all data that a program might need can be stored in the fastest memory. Therefore, operating systems try to predict which data will be needed next. The less likely it is that data will be accessed, the slower the memory can be in which the data is stored.

If the processor needs to access a data set, it first searches in the fastest memory. If the data is not available there, the next slower memory is searched. This is also called cache miss. If the data is found, this is called a cache hit. The concept of storing data in a hierarchy of different fast memories is called a cache hierarchy.

For the estimation of the time a program needs for its execution, it makes a huge difference how many cache misses occur before the data is found with a cache hit. Each cache miss extends the time until the data can be processed by the processor. If this cache hierarchy is not taken into account by the hardware emulator, the time estimates can differ greatly from the execution time on real hardware.

Fig. 3 shows how we implemented this cache hierarchy in the hardware emulator. The processor accesses the memory via a general interface (`MemoryAccess`). Internally, this memory can be implemented by several memories with different timing characteristics. Each of these memories can also have an underlying cache layer that can take over a request if the data sought cannot be found in the current layer. This structure corresponds conceptually to the technical implementation in the computer. The `SetAssociativeMemory` divides the cache into sets of fixed size where each set can contain one or more cache lines. In this context, the *associativity* refers to the number of cache lines a set holds. In case the requested data cannot be found in the cache, the request will be forwarded to the next layer in the cache hierarchy. In contrast, the

`RandomAccessMemory` is modeled using a fixed access time as we consider it the last layer in the cache hierarchy. Additionally, we account for the write-back policies of many cache hierarchies. If write-back is used, new data is not immediately written to main memory. Instead, it is only kept in the cache, *i.e.,* one of the faster parts of the memory. Accordingly, the cache can have a different value than the main memory. To prevent the data from being lost, it needs to be written back to the main memory before invalidating the cache. We add the time for doing this to the total time consumption. Other than marking memory as being different from the main memory, writing to memory is implemented similar to reading from memory in our model.

B. Modeling Instruction Decoding

Processing an instruction by the processor consists of three steps, known as *instruction cycle*: 1) fetch, 2) decode, and 3) execute. During the fetch step, the processor gets the next command to be processed from memory. Then, in the decode phase, the command is interpreted. When using indirect memory addresses, this may require reading the real address from a slower memory and copying it into one of the processor's registers. Additionally, this phase also copies memory that is needed to execute the command to the processor's registers. Then, in the last step, the command is executed. The results of this operation may be written back to memory.

The time needed to execute a command is already represented by the Unicorn emulator that underlies the hardware emulator of [2]. However, the operands of the instruction are ignored even though they can influence the timing. To integrate the operands into the time calculation, we used the ZyDis[2] disassembler library. Using this library, we can get information about the instruction to be executed. Using the instruction latencies from [20], we can estimate the number of CPU cycles needed to execute the decoded instruction. Knowing the processor's clock speed, *i.e.,* the number of CPU cycles per second, we can use this information to estimate the time it takes to execute the command. Fetching the instruction from memory is implemented like accessing the cache hierarchy. Overall, this enables us to estimate the time needed for executing an instruction more accurately.

Alternatively to the emulation mode described above, a live measurement mode can be used. In this case, the autopilot is executed instead of emulated. This has two disadvantages. First, we can only simulate vehicles with the same hardware as our host hardware. Testing the autopilot with other vehicle hardware would require us to execute the simulation on another physical or virtual machine. Second, the results are not reproducible and depend on the current load of the operating system. Recreating exact behaviors can turn out to be difficult.

The measurement mode can be run in several variants. First, we can run the autopilot, measure the duration in each step, and use the measured value directly. Second, we can run the autopilot repeatedly, *e.g.,* 100 times, in each time step and then use the mean value or the empirical WCET. For real WCET analysis tools such as AbsInt aiT can be used.

[2]ZyDis GitHub project. [Online]. Available: `https://github.com/zyantific/zydis` Last accessed: 19.01.2022

V. Evaluation

The research questions to be answered in our evaluation are the following:

RQ1: How well does the proposed emulated memory model represent real hardware?

RQ2: How accurate does the proposed emulation solution approximate the real execution time of a given software?

RQ3: How well does the prediction scale for different applications and different workloads?

RQ4: Is hardware simulation negligible?

The experiments and measurements for the evaluation were executed on a PC with AMD Ryzen 5 3600 CPU (Zen 2 Matisse architecture) and 16 GB DDR4 RAM with 3000 MT/s. This system will be referred to as real hardware in the following. The hardware emulator is configured to emulate the specification of the real hardware. The experiments and measurements on real hardware that are presented in this chapter are based on single-core/thread execution, in order to ensure the same conditions as in the software emulation on the hardware emulator. The evaluation of the hardware emulator as introduced in Section IV is conducted with regard to two aspects. First, the hardware events related to the memory system are collected both on the hardware emulator and real hardware and compared in order to answer RQ1. While hardware events can be logged by the hardware emulator directly, for real hardware we employ the Cachegrind profiler to do this. Second, the execution time of various algorithms is estimated using the hardware emulator and the estimates are compared with runtime measurements on real hardware in order to answer RQ2 and RQ3.

To validate the behavior of the memory system of the hardware emulator, we profiled a neural network training program. To obtain meaningful results, a cache configuration with smaller L1 Data and L1 Instruction cache size than the real hardware was used in this experiment. This is due to the following reason: if the hardware emulator and Cachegrind are configured according to the specification of the real hardware, the L1 Data and L1 Instruction cache are large enough to temporarily save all the data required for the test application. Thus, there won't be enough accesses to L2 and L3 caches since most of the cache accesses into L1 cache will result in cache hits. With such a configuration, it wouldn't be possible to determine whether the cache placement/replacement policy behaves correctly in lower-level caches. Therefore, we decrease the L1 cache size in order to trigger enough L1 cache misses (e.g. L2 cache accesses) to profile the memory system behavior in lower-level caches. The cache configuration for our experiment is shown in Table I.

The absolute numbers of cache operations differ by the factor two approximately, possibly resulting from the dif-

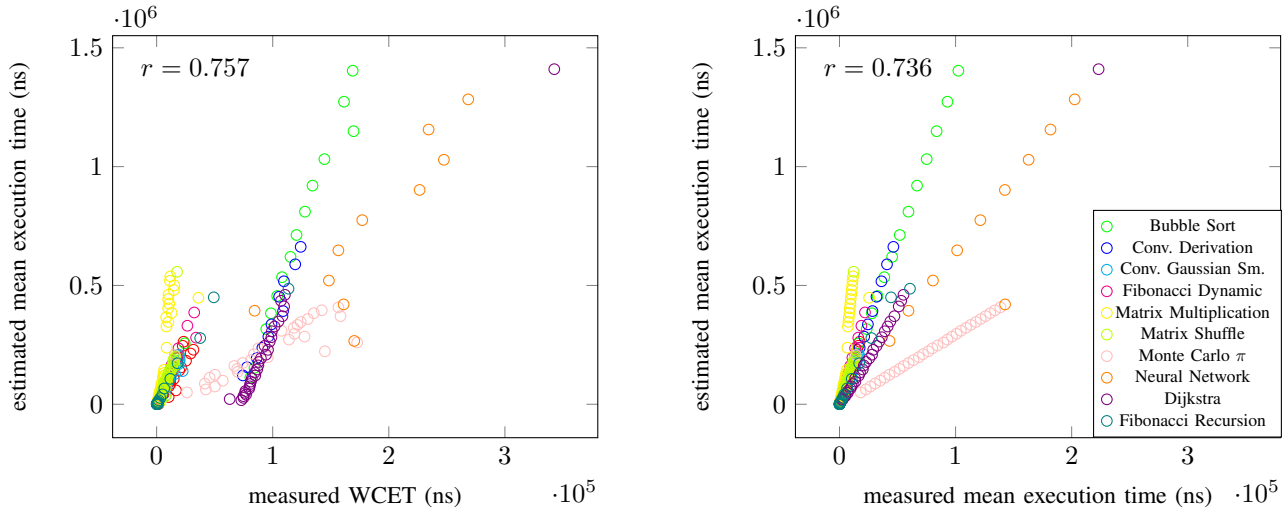

Fig. 4. The two scatter plots illustrate the correlation between measured and estimated execution time for all tested algorithms with various parameters. The horizontal axes denote the measured WCET (top) and the mean execution time, respectively. The Pearson correlation coefficients shown in the top left corners of the two plots measure the degree of linearity between the plotted variables. The legend applies to both plots.

TABLE I

REDUCED CACHE CONFIGURATION FOR THE NEURAL NETWORK PROFILING EXAMPLE.

Level	type	Associativity	Size	Cache Line Size
1	Instruction	2	128 Byte	32 Byte
1	Data	2	128 Byte	32 Byte
2	Shared	8	512 KiB	64 Byte
3	Shared	16	16 MiB	64 Byte

TABLE II

CACHE EVENT COMPARISON BETWEEN THE HARDWARE EMULATOR AND CACHEGRIND.

Hardware Events		HW Emulator	Cachegrind
L1 Inst.	accesses	1	1
	hits	0.755	0.769
	misses	0.245	0.231
L1 Data	accesses	1	1
	hits	0.472	0.452
	misses	0.528	0.548
L2 Shared	accesses	1	1
	hits	0.982	0.997
	misses	0.017	0.003

ferences regarding the execution environment of Cachegrind and the hardware emulator. In order to render the numbers of hits and misses comparable between the hardware emulator and Cachegrind, we normalize the results to the total number of accesses on the respective cache level and platform, i.e. hardware emulator and Cachegrind. The normalized results (i.e. hits and misses per access) are summarized in Table II. The normalized values in each line are similar, indicating that the hardware emulator can imitate the memory behavior of the real hardware remarkably well under equal execution conditions.

To assess the execution time estimation of the hardware emulator and its scalability, we ran experiments on 11 parameterizable algorithms. The parameters were chosen to adjust the complexity and hence produce test programs of different computational loads. The following algorithms were benchmarked: neural network training, Dijkstra, bubble sort, mem bench, Fibonacci, Fibonacci dynamic, convolution derivation, convolution Gaussian smoothing, matrix multiplication, Monte Carlo π, and matrix shuffle. Each algorithm was run 100 times for each parameter in order to estimate the mean, median, and WCET. While the original implementation of the hardware emulator yielded completely unrealistic results several orders of magnitude higher than the measured results, adding instruction context interpretation led to a significant improvement of accuracy. Further adding a memory hierarchy resulted in estimates of the correct order of magnitude when compared to the measured WCET, i.e. the longest execution of the 100 runs for a given algorithm and a fixed parameterization, and measured mean execution time. The results are depicted in the scatter plots in Fig. 4. Ideally, the predicted and the measured values would be the same, hence we use the Pearson coefficient as a measure of linearity to assess the quality of the predicted values (the slope is not relevant as it can be estimated and included into the model). While we observe a very high correlation when looking into one single algorithm (10/11 algorithms have a Pearson coefficient > 0.9), the Pearson coefficient drops to 0.757 and 0.736 for WCET and mean execution time, respectively, if we mix the data. Hence, while the rough estimate is still useful, it seems that some operations are modeled better than others by our emulation concept.

Finally, to illustrate the impact and answer RQ4 we simulate a trajectory following controller with and without

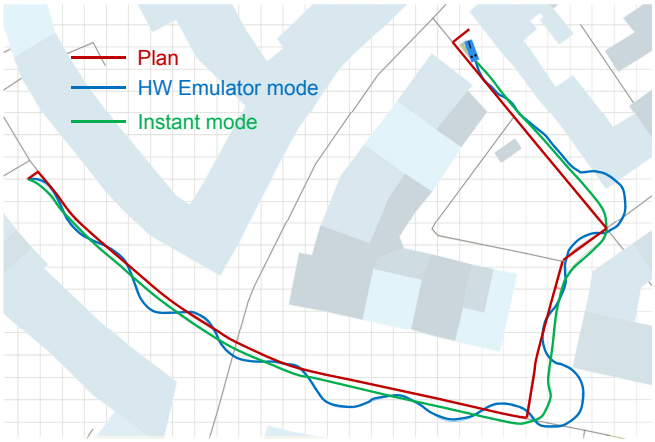

Fig. 5. MontiSim screenshot depicting the planned (red), and driven trajectories of a vehicle. The driven trajectories are depicted in blue and green for simulation with and without hardware emulation, respectively.

hardware emulation. The resulting trajectories for a simple scenario are depicted in Fig. 5. The planned trajectory is depicted in red. The actually driven trajectory is depicted in blue for a simulation run with hardware emulation turned on and in green for a simulation where the results of the autopilot software were available instantly, i.e. neglecting the hardware induced delays. For better readability, the planned trajectory is depicted as the middle of the street, while for the actual trajectories, the controller tries to keep on the right lane. While in instant mode we can observe a smooth driving behavior, the hardware emulation mode trajectory clearly oscillates as the emulated hardware is too slow for the controller[3].

VI. Conclusion

In this paper, we presented a modular, event-driven simulation architecture for intelligent transportation systems with a particular focus on hardware-induced delays. We showed how reproducible simulation results can be obtained for a target hardware platform and how timing estimation can affect the functionality of an autopilot under test. The accuracy of timing estimation was evaluated on a variety of different pieces of code as well as an actual autopilot controller, ensuring a wide coverage of machine instructions. While the estimates did not perfectly match the measurements on real hardware, there is a clear correlation, enabling the developers to take hardware delays into account when developing vehicle software. Two challenges arise in our approach: first, a trade-off between hardware model complexity and accuracy needs to be found. Second, switching to a new hardware platform requires the implementation and evaluation of new hardware models.

[3]To make the effect more pronounced, we added boilerplate code to the relatively simple trajectory keeping controller implementation.

References

[1] I. Drave, T. Greifenberg, S. Hillemacher, S. Kriebel, E. Kusmenko, M. Markthaler, P. Orth, K. S. Salman, J. Richenhagen, B. Rumpe, C. Schulze, M. Wenckstern, and A. Wortmann, "SMArDT modeling for automotive software testing," *Software: Practice and Experience*, vol. 49, no. 2, pp. 301–328, February 2019.

[2] J. C. Kirchhof, E. Kusmenko, J. Meurice, and B. Rumpe, "Simulation of Model Execution for Embedded Systems," in *Proceedings of MODELS 2019. Workshop MLE*. IEEE, September 2019, pp. 331–338.

[3] T. Baumhöfer, W. Waag, and D. Sauer, "Specialized Battery Emulator for Automotive Electrical Systems," in *IEEE Vehicle Power and Propulsion Conference*, 2010, pp. 1–4.

[4] L. Guerrieri, G. Masera, I. S. Stievano, P. Bisaglia, W. R. Garcia Valverde, and M. Concolato, "Automotive Power-Line Communication Channels: Mathematical Characterization and Hardware Emulator," *IEEE Transactions on Industrial Electronics*, vol. 63, no. 5, 2016.

[5] C. Ferdinand, "Worst Case Execution Time Prediction by Static Program Analysis," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, 2004, pp. 125–.

[6] J. Hansen, S. Hissam, and G. A. Moreno, "Statistical-Based WCET Estimation and Validation," in *WCET'09*, N. Holsti, Ed., vol. 10. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2009, pp. 1–11, also published in print by Austrian Computer Society (OCG) with ISBN 978-3-85403-252-6.

[7] Y. Chen and V. Dinavahi, "Hardware Emulation Building Blocks for Real-Time Simulation of Large-Scale Power Grids," *IEEE Transactions on Industrial Informatics*, vol. 10, no. 1, pp. 373–381, 2014.

[8] M. Serror, J. C. Kirchhof, M. Stoffers, K. Wehrle, and J. Gross, "Code-Transparent Discrete Event Simulation for Time-Accurate Wireless Prototyping," in *2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, ser. SIGSIM-PADS '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 161–172.

[9] S.-H. SOng, B.-C. Jeong, H.-I. Lee, J.-J. Kim, J.-H. Oh, and G. Venkataramanan, "Emulation of output characteristics of rotor blades using a hardware-in-loop wind turbine simulator," in *IEEE APEC 2005.*, vol. 3, 2005, pp. 1791–1796 Vol. 3.

[10] W. Deng, Y. H. Lee, and A. Zhao, "Hardware-in-the-loop simulation for autonomous driving," in *34th Annual Conference of IEEE Industrial Electronics*, 2008, pp. 1742–1747.

[11] T. Bokc, M. Maurer, and G. Farber, "Validation of the Vehicle in the Loop (VIL); A milestone for the simulation of driver assistance systems," in *IEEE Intelligent Vehicles Symposium*, 2007, pp. 612–617.

[12] J. C. Kirchhof, E. Kusmenko, B. Rumpe, and H. Zhang, "Simulation as a Service for Cooperative Vehicles," in *Proceedings of MODELS 2019. Workshop MASE*. IEEE, September 2019, pp. 28–37.

[13] F. Grazioli, E. Kusmenko, A. Roth, B. Rumpe, and M. von Wenckstern, "Simulation Framework for Executing Component and Connector Models of Self-Driving Vehicles," in *MODELS 2017. Workshop EXE*, 2017.

[14] C. Frohn, P. Ilov, S. Kriebel, E. Kusmenko, B. Rumpe, and A. Ryndin, "Distributed Simulation of Cooperatively Interacting Vehicles," in *ITSC'18*. IEEE, 2018, pp. 596–601.

[15] E. Kusmenko, A. Roth, B. Rumpe, and M. von Wenckstern, "Modeling Architectures of Cyber-Physical Systems," in *European Conference on Modelling Foundations and Applications (ECMFA'17)*, ser. LNCS 10376. Springer, July 2017, pp. 34–50.

[16] E. Kusmenko, B. Rumpe, S. Schneiders, and M. von Wenckstern, "Highly-Optimizing and Multi-Target Compiler for Embedded System Models: C++ Compiler Toolchain for the Component and Connector Language EmbeddedMontiArc," in *MODELS'18*. ACM, 2018.

[17] E. Kusmenko, S. Nickels, S. Pavlitskaya, B. Rumpe, and T. Timmermanns, "Modeling and Training of Neural Processing Systems," in *MODELS'19*. IEEE, 2019, pp. 283–293.

[18] N. Gatto, E. Kusmenko, and B. Rumpe, "Modeling Deep Reinforcement Learning Based Architectures for Cyber-Physical Systems," in *MODELS 2019. Workshop MDE Intelligence*, September 2019, pp. 196–202.

[19] A. Atouani, J. C. Kirchhof, E. Kusmenko, and B. Rumpe, "Artifact and Reference Models for Generative Machine Learning Frameworks and Build Systems," in *GPCE'21*. ACM SIGPLAN, 2021, pp. 55–68.

[20] A. Fog *et al.*, "Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus," *Copenhagen University College of Engineering*, vol. 93, p. 110, 2011.