VIVE: Virtualization of Vehicular Electronics for System-level Exploration

Md Rafiul Kabir, Neha Mishra, and Sandip Ray

Abstract— **We develop a virtual prototyping infrastructure for modeling and simulation of automotive systems. We focus on exercising and exploring use cases involving system-level coordination of vehicular electronics, sensors, and software. In current practice, such use cases can only be explored late in the design when all the relevant hardware components are available. Any design change, *e.g.*, for optimization or security or even functional errors found during the exploration, incurs prohibitive cost at that stage. Our solution is a flexible, configurable prototyping platform that enables the user to seamlessly add new system-level use cases. Unlike other related prototyping environments, the focus of our platform is on communication and coordination among different components, not the computation of individual Electronic Control Units. We report on the use of the platform for implementing several realistic usage scenarios on automotive platforms and exploring the effects of their interaction. In particular, we show how to use the platform to develop real-time in-vehicle communication optimizers for different optimization targets.**

I. INTRODUCTION

Vehicular systems have seen a rapid transformation in recent years, with the infusion of autonomous features targeted to augment and in many cases, replace human operation. Autonomous features hold the promise of dramatically increasing safety by reducing human errors, while concurrently enabling efficient utilization of the transportation infrastructure and minimizing environmental impacts. However, one upshot of this trend is an explosion in electronic and software complexity. A modern automotive system can contain hundreds of electronic control units (ECUs) each connected to a number of sensors and actuators, multiple in-vehicle networks, and several hundred megabytes of software code. Unsurprisingly, these systems can have subtle, hard-to-detect errors. Many of these errors involve concurrent coordination of multiple components: an "innocent" optimization in one component (*e.g.*, how a specific ECU handles a message coming from the CAN bus) can have an unpredictable effect on coordination, timing, performance, or even functional impact that can compromise reliability, safety, and efficiency; furthermore, bugs that escape to deployment can be exploited in the field by adversaries to cause accidents, introduce inefficiency in transportation, or even a breakdown of the transportation infrastructure.

In current industrial practice, exploration of full-system functionalities is performed by OEM through *field testing*. This entails connecting all the electronic components, sensors, and actuators (possibly excluding the mechanical chassis), installing the appropriate software, and physically exercising the different vehicular use cases (*e.g.*, pressing the brake, turning the steering wheel, etc.). While such field testing enables systematic exploration of system-level use cases, a key problem with this approach is that it can only be performed late in the design, when all the hardware components, sensors, and actuators have been manufactured and assembled. At this stage, any substantial re-design is generally avoided since it can result in significant churn in the design and production timeline. Consequently, any non-essential optimization has to be eschewed. Even design errors found this late can be expensive to address. There is obviously a critical need for a platform that will enable exploration and validation of system-level functionalities early in the design.

In this paper, we develop a platform to address this crucial challenge. Our solution is a prototype environment, VIVE, that is targeted to enable simulation and exploration of automotive system-level use cases, help comprehend their interactions, and facilitate system-level optimization and security, present the system architecture for VIVE, and discuss the extensibility and configurability features necessary to make the system viable. We have used VIVE to implement a number of representative vehicular use cases. We also demonstrate the utility of the framework in designing a real-time communication packet optimizer for the CAN bus.

The remainder of the paper is organized as follows. Section II provides the relevant background and summarizes related research in this area. Section III presents the VIVE platform architecture and its use in vehicular use case implementations. In Section IV we discuss the use cases implemented. To illustrate the use of the platform, we go into detail of one use case for VIVE, *viz.*, the Antilock Breaking System (ABS). In Section V we discuss the application of the platform in system-level optimization. We conclude in Section VI.

II. RELATED WORK

A. Virtual Prototyping

Virtual Prototyping entails developing a software (*i.e.*, virtual) model of a design before a physical embodiment is built. As systems get increasingly complex, prototyping is getting increasingly common to facilitate early exploration and optimization. A popular prototyping approach makes use of *digital twins* [1]. The idea is to create a digital representation of a physical device or object that mimics the underlying

*This research has been supported in part by the National Science Foundation under Grant No. CNS-1908549.

Md Rafiul Kabir, Neha Mishra, and Sandip Ray are with the Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32611, USA. Email: kabirm@ufl.edu, nehamishra@ufl.edu, sandip@ece.ufl.edu.

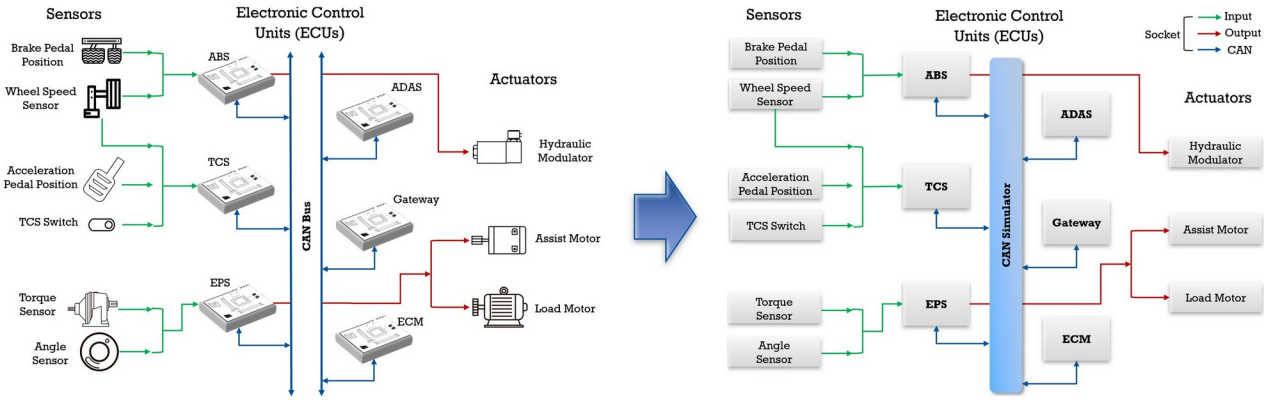

Fig. 1. Virtualization of Use Cases. **(Left)** A typical physical prototype to support three use cases: Antilock Braking System, Traction Control System, and Right Turn. **(Right)** Corresponding Virtualization.

physics and use that data to develop a mathematical model that simulates the real-world in digital space. Digital twins have been used for a number of applications, including space capsules, factories, and smart cities [2]. However, the focus has been primarily on modeling physical and mechanical behavior.

The idea of prototyping has also become popular in the "cyber" world, particularly in the area of digital hardware and System-on-Chip (SoC) designs. Such platforms are referred to as *virtual platforms*. The idea is to develop a software model of the underlying hardware platform. The idea is to provide a platform early in the design (before silicon or even mature register-transfer level (RTL) models are available) which can be used for design exploration and software development. Obviously, many hardware corner-cases are abstracted in the virtual model. Nevertheless, the approach has been found valuable for early software and firmware development, hardware-software co-design, and co-validation [3], [4]. A number of frameworks have been developed to enable virtual prototyping, including several commercial offerings [5].

Our work derives inspiration from the progress above in digital twins as well as virtual platforms and targets them towards developing a focused prototyping solution for automotive systems. Analogous to digital twins, we target the overall system functionality, not on a specific electronic component. However, unlike digital twins, — and analogous to virtual platforms, — the focus is not on physical or mechanical behavior but on the cyber components of the system. In fact we focus primarily on two aspects of the cyber behavior: (1) software functionality and (2) coordination and communication of the components through CAN messages. Furthermore, we develop a generic and extensible platform that can be integrated with a variety of use cases. To our knowledge, there is no other previous work that enables automotive virtual prototyping at this level.

B. Virtualization of automotive components

There has been some recent work to virtualize automotive components. Strobl *et al.* [6] provide a comprehensive discussion on the benefits of automotive virtualization as a foundation for consolidating a multitude of ECUs into a few Domain Controller Units (DCUs) Lee *et al.* [7] developed a Virtualized Automotive Display System which can manage multiple execution domains like the automotive control software and the in-vehicle infotainment (IVI) software. Safar *et al.* [8] added a VP to the V-model of automotive software development as part of an enhanced methodology that allows verification and validation on SoC, ECU, and system level. It also provides fault injection capability and co-debugging mechanism of AUTOSAR software. While these approaches cover virtualization of some components, the framework unlike ours only considers specific subsystems. *Our approach is complementary to these activities, focusing instead on system-level aspects while abstracting individual ECUs and subsystems. While other automotive simulators like CARLA and SUMO focuses on certain features of autonomous vehicle controls and traffic simulations respectively,* VIVE *stands differently by providing the extensibility of new use-cases, explore the inter component and system interactions and, exercise optimization and security targets.*

III. VIVE Architecture and Implementation

A. High-level Architecture

An automotive system comprises ECUs, possibly with sensors and actuators attached, together with in-vehicle networks (*e.g.*, CAN). Any automotive use case (*e.g.*, right turn, brake, etc.) is initiated by some actuarial action and involves a sequence of messages transmitted across the network by different ECUs. Fig. 1 illustrates the virtualization involved in representative use cases. Given this insight, VIVE enables modeling different use cases by providing the following infrastructure.

- **Network Simulator:** VIVE provides a simulation environment for the in-vehicle networks involved in the use case. The network simulator faithfully models the protocols implemented (*e.g.*, CAN, LIN, etc.).
- **ECU:** Each ECU is, of course, a complex computational element. Unlike traditional virtual platforms, VIVE does not require a complete software model of the ECU. Instead, it provides a generic interface through

TABLE I

SOME USE CASES IMPLEMENTED AND CORRESPONDING VEHICULAR COMPONENTS

Name	ECU	Sensor	Actuator
Anti-lock Braking System (ABS)	ABS, ADAS, Gateway	Wheel Speed Sensor, Brake Pedal Position	Hydraulic Modulator
Right Turn	Electric Power Steering (EPS), ABS, Gateway	Angle Sensor, Torque Sensor	Assist Motor, Load Motor
Return-to-Center	EPS, ABS, Gateway	Angle Sensor, Torque Sensor	Assist Motor
Traction Control System (TCS)	Engine Control Module (ECM), TCS, ADAS, Gateway	TCS Switch, Wheel Speed Sensor, Acceleration Pedal Position	
Cruise Control	Body Control Module (BCM), Gateway, Engine Control Module (ECM)	Cruise Switch, Acceleration Pedal Position, Acc/Dec Switch, Wheel Speed Sensor	Throttle
Indirect Tire Pressure Monitoring System (iTPMS)	ABS, Gateway	Wheel Speed Sensor	
Direct Tire Pressure Monitoring System (dTPMS)	Tire Pressure Monitoring System (TPMS), Gateway	Tire Pressure Sensor	

which one can connect either (1) an actual ECU, or (2) a simple hardware platform (*e.g.*, Raspberry Pi) to simulate the functionality of the ECU as relevant to the use cases, or (3) a software implementation of the algorithm (see below).

- **Sensors/Actuators:** Analogous to the ECUs, VIVE provides an interface which can be used to connect a physical sensor/actuator or simply a software process generating synthetic sensory or actuarial data.

Our design choices above stem from the goal of the platform to enable exploration and optimization of the system-level coordination involved in different use cases and their interactions. In particular, the vehicular communications consequently become the centerpiece of the system while the details of ECUs and sensors (beyond the functionality required to comprehend the use case) can be abstracted.

B. ECU, Sensor, and Actuator Models

The goal of VIVE is to enable the user to get a realistic idea of the impact of different use cases that interact with one another. Table I shows the list of vehicular components for some of the implemented use cases. A use case is generally initiated by an actuarial activity, *e.g.*, the ABS will be initiated by a user pressing the brake pedal. However, other components (*e.g.*, sensors and many ECU computations) relevant to the use case perform continuous, ongoing activity, *e.g.*, the wheel speed sensor, although relevant to ABS, continues ongoing activity independent of the actuarial actions initiating the use case. VIVE supports this duality as follows. Sensors (or processes simulating the sensor actions) are sampled continuously. Correspondingly, all computation blocks keep running continuously. For example, the wheel speed sensor implemented provides simulated wheel speed data continuously giving inputs to the ECU and the brake pedal position sensor gets the actual input from the user pressing a simulated brake pedal.

Note that in some cases the actuator takes input from the ECU to perform certain mechanisms. The actuator simulation process in VIVE consequently supports the communication of inputs from the simulated (or actual) ECU and provides output data with the final result of the mechanism. For example, the simulated assist motor takes input from the simulated Electric Power Steering (EPS) ECU and provides the necessary output with a result of the ECU computation for the next mechanism. This output indicates how much assist torque is being applied to the steering wheel for the right turn. Finally, the (simulated) ECU is connected and interfaced with the simulated in-vehicle network (see below) to interact with other ECUs. Unlike sensors and actuators, the ECU process can both receive and send messages. Furthermore, ECUs can take part in multiple use-cases with necessary computation as needed, *e.g.*, note from Table I, ABS ECU is involved in four use cases.

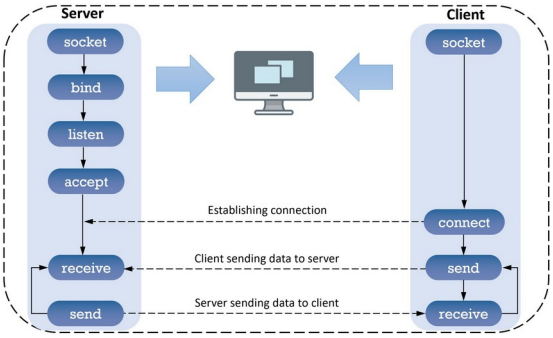

Fig. 2. TCP client-server socket flow

We end the discussion on the vehicular components with a note on the versatility of the VIVE platform. Recall that VIVE enables ECUs to be simulated with either indigenous processes or by connecting an actual hardware ECU or through a configurable microcontroller such as Raspberry Pi. The third component (Raspberry Pi) is an interesting aspect of the platform. It enables the user to simulate the functionality of an ECU and execute (and validate) real software, without requiring the availability of a full-blown ECU hardware as necessary for field testing.

C. Communication

As mentioned earlier, exercising communications among systems is the central goal of VIVE. There are two types of communication shown: (1) Direct Electrical signal, and (2) Communications through in-vehicle networks. VIVE

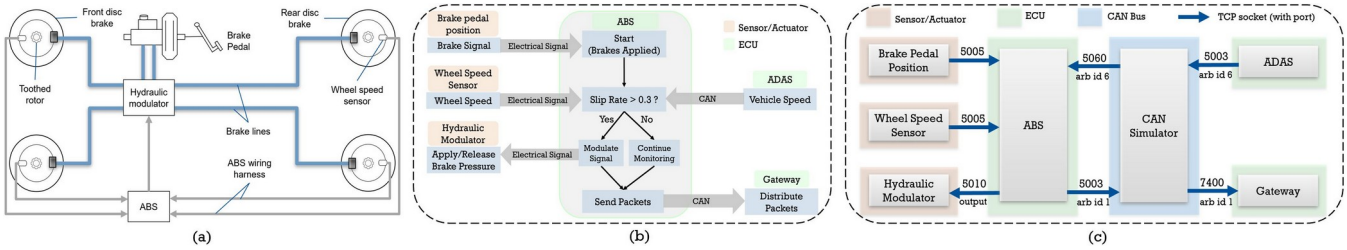

Fig. 3. Anti-lock braking system (a) automotive hardware architecture (b) platform's functionality flow diagram and (b) platform's primary system design

currently supports CAN communication. The CAN simulator uses the arbitration id from the CAN frame to determine which ECU should receive data from it. Each arbitration id is tied with a port number that denotes a specific ECU. The simulator receives all the CAN messages via socket then sends that CAN message to the corresponding ECU. The simulator can also handle multiple messages from multiple ECUs of various use-cases at the same time.

Remark 1: (Implementation Note) V IVE implementation of communication uses Transmission Control Protocol (TCP) sockets to realize interaction among all the processes. This socket programming is based on the client-server model shown in Fig. 2. The client typically initiates the communication while the server waits passively to respond to the client's request [9]. For the implementation of the CAN bus, a CAN simulator is modeled which uses a CAN library to build the CAN frame. This frame is constructed as a byte array message to send and receive via the socket. The message contains arbitration id, extended id, data length, and the actual data. The rest of the communication with sensors and actuators, *i.e.*, electrical signals is simulated as a simple byte-array message without any CAN frame.

D. Extensibility and User Interface

A key feature of VIVE is support for *extensibility*, *i.e.*, seamless introduction of new use cases (including those that possibly interact with the existing ones). To extend V IVE with a new use case, the user needs to provide the following information:

1) ECUs involved, and the computation being performed by each ECU to realize the use case.
2) Sensors and actuators involved. If a new sensor is necessary for the use case then the sensor process needs to be added to the system.
3) Message sequences communicated by the different ECUs (and the modes of communication, *e.g.*, via CAN, direct electrical signals, etc.).

The platform manages interaction among use cases, scheduling, and resource sharing. For instance, in Fig. 1, the wheel speed sensor takes part in both ABS and TCS use cases. The platform provides a graphical user interface (GUI) to perform actuarial actions such as pressing the brake. Finally, during simulation, VIVE permits the user to specify any subset of the supported use cases and study their interactions.

TABLE II
OUTPUT BYTE ARRAY MESSAGES

Component	Message
Brake Pedal Position	[0] or [1]
Wheel Speed Sensor	simulated values from [0] to [60]
ADAS	simulated values from [0] to [60] (CAN)
ABS (final output)	[0] or [1000] (CAN)
ABS (for gateway)	[0] or [1] (CAN)

IV. USE CASES

A. Use Case Summary

Table III presents the summary of use cases implemented with VIVE. Note that each use case involves multiple ECUs, sensors, and actuators. Furthermore, sensors and ECUs are shared among the use cases (*e.g.*, ABS ECU and wheel speed sensors). The range and diversity of use cases supported demonstrates the flexibility of VIVE and its viability as a prototyping platform.[1]

B. An Illustrative Use Case: ABS Implementation on V IVE

To enable a better understanding of the V IVE, we now take a closer look at the implementation of ABS as an illustrative example. Fig. 3 provides an overview of this use case and V IVE implementation. The functionality is fairly standard [10]–[12]. The simulated components include indigenous processes for the ECUs, and sensors, *e.g.*, ABS ECU, ADAS ECU, gateway, hydraulic modulator, brake pedal position sensor, and wheel speed sensor. The V IVE implementation uses sockets for sending data among these processes as discussed in Section III-C. The actuarial action from the user initiating the use case is the pressing of the brake, which is enabled through the V IVE GUI. On the action, the brake pedal position sensor sends the brake position signal to the ABS ECU. The slip rate is calculated from the simulated speed values of the wheel speed sensor and ADAS. Table II shows all the byte array messages sent by the components.

If the slip rate is above 0.3, then the ABS sends a byte array message to the hydraulic modulator to apply or release brake pressure. Since the parameters for brake pressure vary

[1] The functionality of the use cases has been currently implemented till the gateway to focus on the driving and driving assistance operations; the operations of the instrument cluster (*e.g.*, notifications in heads-up display) are not implemented. However, from the platform perspective the instrument cluster functionality can be implemented in the same way as the driving assistance operations.

TABLE III
IMPLEMENTED USE -CASES

Use-case	Functionality
Anti-lock Braking System (ABS)	Activates ABS while braking when the wheels are locked. The ABS ECU takes multiple inputs to compute ABS activation. The vehicle speed data is sent as CAN frame from ADAS to ABS and in response, ABS sends the ABS status as CAN message to gateway.
Right Turn	User makes a right turn simulated by angle sensor and torque sensor of the steering wheel. The EPS ECU computes the assist and load torque required. The vehicle speed is sent as CAN from ABS to EPS and in response, EPS sends the turn status as CAN message to gateway.
Return-to-Center (RTC)	After applying right turn, the steering wheel returns to the center simulated by the angle sensor and torque sensor. The EPS ECU computes the RTC assist torque required. The vehicle speed is sent as CAN frame from ABS to EPS and in response, EPS sends the RTC status as CAN frame to gateway.
Traction Control System (TCS)	Activates TCS while accelerating. The TCS ECU takes multiple sensor inputs to compute TCS activation. The vehicle speed is sent as CAN frame from ADAS to TCS and in response, TCS sends the torque reduction status as CAN message to ECM and gateway.
Cruise Control	User activates the cruise control to increase, decrease or maintain the vehicle speed. The ECM ECU takes sensor inputs and computes the speed direction. User input data is sent as CAN from BCM to ECM and in response, ECM sends the cruise status as CAN message to gateway.
Indirect Tire Pressure Monitoring System (iTPMS)	ABS ECU calculates tire pressure from the wheel speed sensor to check pressure status. Tire pressure warning data is sent as CAN message from ABS to gateway.
Direct Tire Pressure Monitoring System (dTPMS)	TPMS ECU directly calculates tire pressure from the tire pressure sensor to check pressure status. Tire pressure warning data is sent as CAN message from TPMS to gateway

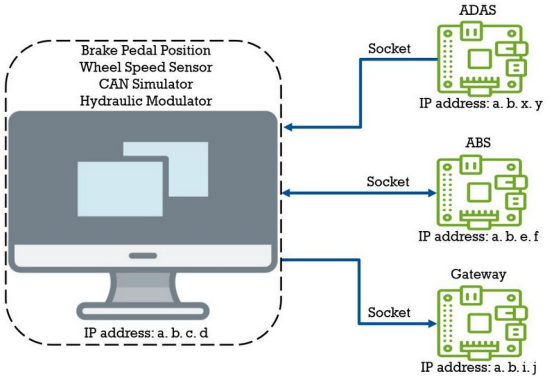

Fig. 4. Raspberry Pi integration for ABS

between 1000 psi and 1600 psi in a vehicle [13], the byte array message sent to the hydraulic modulator is simulated as `[0]` for no pressure and `[1000]` for pressure application. Since this is a continuously running process, the slip rate keeps changing based on different speed data resulting in two different final outputs (ABS activated or not activated). Additionally, the ABS ECU sends the CAN frame back to the CAN simulator with an arbitration id 1, which denotes the gateway ECU.

Finally, note that the ABS use case can also be implemented with Raspberry Pi models for the respective ECUs to enable exploration of functionality with real software. Fig. 4 shows the Raspberry Pi Integration for ABS use case. Here all the processes for the sensors, actuators, and CAN bus are implemented as separately while ADAS, ABS, and gateway ECUs are realized through RPIs. For this integration, all four machines will have their host IP address to connect thus building an embedded system with more computation ability. Since the RPI is capable of integrating with actual physical sensors, VIVE also permits the integration of physical sensors instead of indigenous processes.

V. OPTIMIZATION USING VIVE

One key application of the VIVE platform is early system-level optimization. To showcase this ability, we use VIVE to implement two (independent) real-time CAN packet scheduling optimizers: (1) to minimize CAN bandwidth, *i.e.*, number of CAN packets concurrently on the fly); and (2) to minimize latency, *i.e.*, the average time taken for a CAN packet from initiation at source ECU to its delivery at the destination ECU. The goal here is *not* to develop a high-quality optimization algorithm but only to demonstrate how VIVE enables one to design such a real-time optimizer. Note that in current practice since the system-level scenarios like the ones discussed in this paper are only exercised late during field testing, optimizing packet scheduling has to be done offline without accounting for the interactions of different use cases: with VIVE, the user can observe the effects of this interaction and sharpen the optimizer.

For our real-time optimizers, we use Simulated Annealing [14]. Algorithm 1 provides a high-level overview of the implementation. Simulated Annealing provides an approximation to the optimal by formalizing the notion of *slow cooling* in metallurgy as a slow decrease in the probability of accepting worse solutions as the solution space is explored. In our real-time implementation, once the final temperature is reached and the approximate sequence is calculated, the CAN bus transmits the packets mentioned in the first cycle of the sequence, and the rest of the packets are delayed to the next cycle of the bus.

Fig. 5 shows the results of scheduling with simulated annealing optimizers for four interacting use cases. Obviously, the optimization values depend on the use cases considered and the absolute values may differ for other implementations and other use case combinations. However, the graphs show the value of the platform in practice for real-time optimizations. Note that for the use cases considered, scheduling CAN packets to optimize for latency shows a significant impact

Algorithm 1 Algorithm for optimization with congestion

```
while N cycles of CAN bus do
    Receive packets
    Create Default Sequence as per the latency constraints
    while Temp and Iter == Final Temp and Iter do
        Randomly select a sequence while reducing T
        Calculate Peak and Average congestion
        Compare current congestion with least congestion
        if Accept == True then
            Replace least congestion with least congestion
        else if Compare with random number then
            Save with other probability
        else
            Discard the packet
    Send packets in current cycle and hold other packets
```

on the outcome but optimization for congestion seems to have little effect.[2] Such insights, if obtained early in design exploration, obviously enable systematic and efficient CAN scheduling.

VI. Conclusion

We have developed a prototyping platform VIVE for systematically exercising vehicular system-level use cases. VIVE is a flexible, configurable platform infrastructure that enables specification of use cases with components at various levels of abstraction, *e.g.*, as indigenous processes, real hardware, or microcontrollers. Furthermore, the platform enables smooth extension with new use cases. We implemented a variety of use cases with VIVE and also demonstrated the value of VIVE in developing a real-time scheduling optimizer.

In future work, we plan to extend VIVE with new use cases and implement other optimization algorithms. We will also explore the use of VIVE in the early exploration of functional safety and security properties.

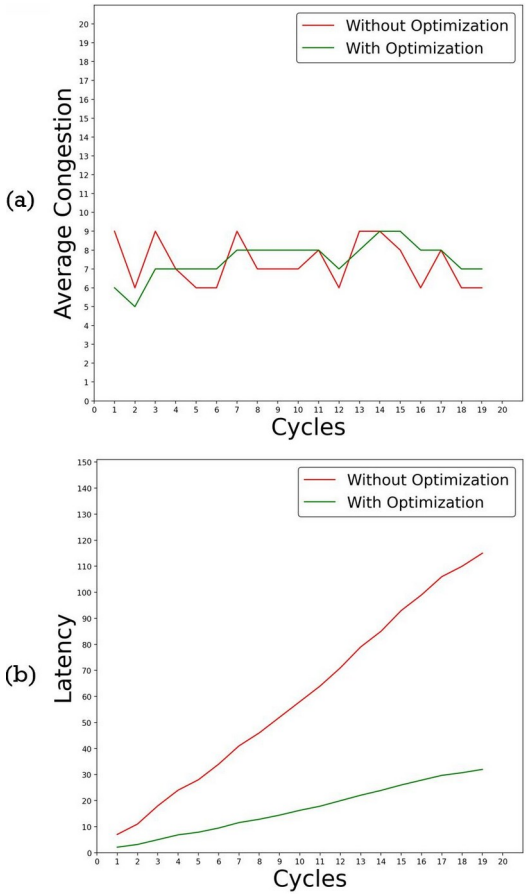

Fig. 5. Results of Simulated Annealing implemented on VIVE. Use cases exercised by the scheduling optimizer are ABS, TCS, Right Turn, and Return-to-Center. (a) Congestion vs. Time. (b) Latency vs. Time.

References

[1] D. Wagg, K. Worden, R. Barthorpe, and P. Gardner, "Digital twins: State-of-the-art and future directions for modeling and simulation in engineering dynamics applications," *ASCE-ASME J Risk and Uncert in Engrg Sys Part B Mech Engrg*, vol. 6, no. 3, 2020.

[2] D. Hartmann and H. van der Herweraer, "Digital Twins," in *Progress in Industrial Mathematics: Success Stories*, 2021, pp. 3–18.

[3] S. Ahn and S. Malik, "Automated firmware testing using firmware-hardware interaction patterns," in *CODES+ISSS*, 2014, pp. 25:1–25:10.

[4] R. Kannavara, , C. J. Havlicek, B. Chen, M. R. Tuttle, K. Cong, S. Ray, and F. Xie, "Challenges and Opportunities in Concolic Testing," in *Proceedings of the National Aerospace Electronics Conference – Ohio Innovation Summit*, 2015.

[5] Synopsys, "Virtualizer," https://www.synopsys.com/verification/virtual-prototyping/virtualizer.html, 2019.

[6] M. Strobl, M. Kucera, A. Foeldi, T. Waas, N. Balbierer, and C. Hilbert, "Towards automotive virtualization," in *2013 International Conference on Applied Electronics*. IEEE, 2013, pp. 1–6.

[7] C. Lee, S.-W. Kim, and C. Yoo, "Vadi: Gpu virtualization for an automotive platform," *IEEE Transactions on Industrial Informatics*, vol. 12, no. 1, pp. 277–290, 2015.

[8] M. Safar, M. A. El-Moursy, M. Abdelsalam, A. Bakr, K. Khalil, and A. Salem, "Virtual verification and validation of automotive system," *Journal of Circuits, Systems and Computers*, vol. 28, no. 04, p. 1950071, 2019.

[9] R. L. R. Maata, R. Cordova, B. Sudramurthy, and A. Halibas, "Design and implementation of client-server based application using socket programming in a distributed computing environment," in *2017 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC)*. IEEE, 2017, pp. 1–4.

[10] V. D. Gowda, A. Ramachandra, M. Thippeswamy, C. Pandurangappa, and P. R. Naidu, "Modelling and performance evaluation of anti-lock braking system," *J. Eng. Sci. Technol*, vol. 14, no. 5, pp. 3028–3045, 2019.

[11] T. Matsushita, K. Kondo, T. Yasuda, and H. Watanabe, "Abs control unit," *Fujitsu Ten Tech*, vol. 6, pp. 52–62, 1994.

[12] A. A. Aly, E.-S. Zeidan, A. Hamed, F. Salem, *et al.*, "An antilock-braking systems (abs) control: A technical review," *Intelligent control and Automation*, vol. 2, no. 03, p. 186, 2011.

[13] ASE, "Abs accumulators," 2008. [Online]. Available: https://www.freeasestudyguides.com/abs-pump-accumulator.html

[14] K. Dharageshwari and C. Nayanatara, "Multiobjective optimal placement of multiple distributed generations in ieee 33 bus radial system using simulated annealing," in *2015 International Conference on Circuits, Power and Computing Technologies [ICCPCT-2015]*, 2015, pp. 1–7.

[2] The reason optimization for congestion has little effect and even has a worse performance at some cycles than non-optimized scheduling is that the number of packets delayed from cycle t to cycle $t+1$ to conserve bandwidth at cycle t is greater than the number of new packets received by the bus at cycle $t+1$.