# Code-transparent Discrete Event Simulation for Time-accurate Wireless Prototyping

Martin Serror[‡], Jörg Christian Kirchhof[‡], Mirko Stoffers[‡], Klaus Wehrle[‡], James Gross[§]
[‡]Communication and Distributed Systems, RWTH Aachen University, Germany
[§]School of Electrical Engineering, KTH Royal Institute of Technology, Sweden
{serror, kirchhof, stoffers, wehrle}@comsys.rwth-aachen.de, james.gross@ee.kth.se

## ABSTRACT

Exhaustive testing of wireless communication protocols on prototypical hardware is costly and time-consuming. An alternative approach is network simulation, which, however, often strongly abstracts from the actual hardware. Especially in the wireless domain, such abstractions often lead to inaccurate simulation results. Therefore, we propose a code-transparent discrete event simulator that enables a direct simulation of existing code for wireless prototypes. With a focus on lower layers of the communication stack, we enable a parametrization of the simulation timings based on real-world measurements to increase the simulation accuracy. Our evaluation shows that we achieve close results for throughput (deviation below 3 % for UDP) and latency (corrected deviation about 13 %) compared to real-world setups, while providing the benefits of code-transparent simulation, i.e., to flexibly simulate large topologies with existing prototype code. Moreover, we demonstrate that our approach finds implementation defects in existing hardware prototype software, which are otherwise difficult to track down in real deployments.

## CCS Concepts

•**Networks** → *Network simulations;* •**Hardware** → *Wireless devices; Testing with distributed and parallel systems;*

## Keywords

Wireless Communications; Discrete Event Simulation; Hardware Prototypes; Testing Methods

## 1. INTRODUCTION

For testing and development of wireless communication protocols, a combined methodology of computer simulation and prototypical hardware deployments, which facilitates the transition from one domain into the other, is highly beneficial. Regarding prototypical hardware, so-called Software Defined Radio (SDR) platforms emerged [25] offering an open and flexible way to modify hard- and software, while enabling testing under realistic conditions. Such a platforms typically consist of a Field Programmable Gate Array (FPGA), a radio interface, and several I/O ports. Thus, they enable a flexible implementation of custom-made physical and data link layer protocols and are particularly suited for the development and testing of new wireless protocols. It is, however, costly to deploy and maintain complex topologies of such devices, which often impedes developers and researchers to exhaustively test their protocols in larger distributed setups.

In contrast, a relatively cheap and powerful method to test and evaluate new communication protocols is by means of simulation. Computer simulation allows to perfectly control a simulated environment and thus to ensure reproducibility [5], which enables testing of specific aspects of a newly proposed protocol. Moreover, it facilitates the discovery of implementation defects, which is especially useful in distributed settings, offering a global view on the current protocol state of each simulated instance. Advanced simulation tools using the Discrete Event Simulation (DES) paradigm ensure scalability and flexibility of the simulated communication scenarios. Prominent examples for such tools are ns-3 [24] and OMNeT++ [1, 26]. Nevertheless, at a certain point, the simulation abstracts from the real world, which might, depending on the use-case, lead to inaccurate behavior in the simulation.

In general, the development of a communication protocol benefits from both simulation and prototypical implementation, as the advantages of both methods nicely complement each other: Simulation enables the evaluation of large-scale scenarios and facilitates debugging, while prototypical implementation offers a realistic view on possible side effects. In practice, however, this implies that two models of the protocol specification are needed, i.e., one for the prototypical hardware and one for the simulation. As each model typically follows its own paradigm, the models can not be easily translated into each other. This leads to two different models, where, in general, it is hard to show that the models are equivalent to each other and thus impedes a direct comparison of both.

For ns-3, Lacage proposed Direct Code Execution (DCE) [14] to execute existing user space as well as kernel space protocol implementations in the simulator. This extension enables a seamless transition from real network deployments to computer simulation and further improves the comparability to the real world. However, DCE operates independently of soft- and hardware processing times, and therefore only offers limited performance insights, especially when considering prototypical hardware deployments.

In this paper, we propose *Code-transparent Wireless Prototype Simulation (CoWS)*, a DES approach built upon ns-3 and DCE to comprehensively develop and test communication protocols both on real wireless prototyping hardware and in simulations. In this context, we refer to code-transparency as executing the same, unmodified user code on hardware boards and in simulations. This eliminates the risk of errors that might occur when translating from hardware implementation to simulation model and thereby enables a

*SIGSIM-PADS'17, May 24 - 26, 2017, Singapore, Singapore*

ISBN 978-1-4503-4489-0/17/05. . . $15.00

DOI: http://dx.doi.org/10.1145/3064911.3064913

seamless cyclic development process including prototypical deployment and simulation. The code-transparent approach thus allows to establish a direct relation between real world and simulation and is, therefore, an important step towards achieving realistic and reproducible simulation of wireless communication protocols and network applications. Furthermore, when carefully calibrating the DES, based on measurements obtained from an SDR platform, accurate results can be achieved in both domains. Finally, we show that CoWS, through its inherent debug capabilities, eases the process of finding implementation defects compared to debugging in distributed hardware deployments. Our approach thus contributes to short development cycles for new wireless protocols enabling reliable and well-tested software for researchers and developers.

In particular, our contributions are as follows.

- A thorough analysis of the necessary steps to enable code-transparent simulation (see Sec. 2);
- A detailed design description of Code-transparent Wireless Prototype Simulation (CoWS) to show the feasibility of our code-transparent DES approach (see Sec. 3);
- A validation of our methodology (see Sec. 4) and a performance evaluation of CoWS showing its applicability in scalable scenarios (see Sec. 5); and
- A case study illustrating new debug possibilities (see Sec. 6).

The paper concludes with a related work discussion (see Sec. 7) and some final remarks.

# 2. ANALYSIS

The goal of this paper is to propose a *code-transparent* approach to combine the advantages of experimentation on real-world SDRs with experimentation in scalable DES. We define code-transparency as the property that the same user code, which runs on hardware devices, also runs in the simulation. In this section, we describe the preliminary steps that are needed to integrate a hardware design into a simulator, following a DES paradigm.

In general, prototypical hardware boards, such as SDRs, consist of *hardware components*, e.g., a microprocessor, and a *software library* that allows accessing the hardware components. On top resides the *user code*, which implements certain functionality based on the underlying hardware. The library thus allows the *user code* to easily access existing hardware, e.g., sending a packet over the radio, without having to deal with low-level hardware details.

One of the most challenging tasks in simulation is to find the right level of abstraction, trading accuracy for complexity [29]. Although most analytic considerations in this section are not limited to a specific SDR design, we base our analysis on the 802.11 Reference Design for Wireless Open Access Research Platform (WARP) v3 [17] to provide a concrete example. WARP is widely used in wireless communications research and is part of many scientific publications in this area, e.g., [2, 6, 30]. In this section, after providing a short introduction to the general architecture of the 802.11 Reference Design, we analyze the different aspects that need to be considered in a code-transparent DES. Afterward, in Sec. 3, we describe the design of Code-transparent Wireless Prototype Simulation (CoWS), which is based on the analysis.

## 2.1 The WARP 802.11 Reference Design

The 802.11 Reference Design provides an implementation of the IEEE 802.11-2012 standard [10], where parts of the standard are realized in hardware, i.e., with the help of an FPGA, while other parts are realized in software, i.e., running on two Xilinx MicroBlaze Central Processing Units (CPUs). The basic architecture of the 802.11 Reference Design is depicted in Fig. 1.

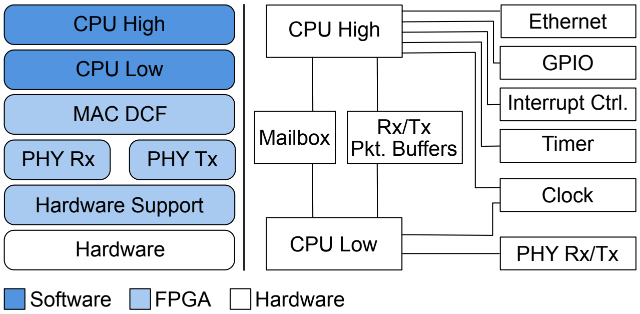


**Figure 1: Simplified architecture of the WARP v3 802.11 Reference Design (adapted from [20]). On the left, the different layers of the architecture are shown. On the right, the access to hardware components from the user code, i.e., CPU High and CPU Low, is illustrated.**

The architecture consists of WARP v3 hardware, custom FPGA cores, and two Xilinx MicroBlaze CPUs, i.e., CPU High and CPU Low. The user code, running on the Xilinx MicroBlaze CPUs, implements major functionalities of the Medium Access Control (MAC) protocol. It further parameterizes the FPGA core, where parts of the MAC and Physical (PHY) protocol are realized, and thus allows to flexibly configure its components. The interface between hardware and software is a library offering a set of functions to the user code, which, depending on their task, set certain registers or manipulate the hardware in another way.

For our code-transparent simulation, this library offers a straightforward abstraction layer from real world to simulation. Instead of manipulating real hardware, we change the library functions to translate their functionality into DES. At the same time, the function prototypes remain unchanged to enable code-transparent simulation of the user code. In the following, we analyze the different components of embedded system designs and explain the required adaption to match DES.

## 2.2 Interrupts and Polling

Basically, there are two different ways for a processor to notice changes in attached hardware components: Either by interrupting the current execution with the help of an interrupt controller or by continuously polling the components for changes. Both methods should be supported by the simulation engine.

Interrupts nicely match the DES paradigm, as an interrupt simply implies adding a new event into the future event set. The event handler can process this event like any other event, i.e., with no additional overhead. However, a polling approach requires a bit more logic: When the user code actively polls for changes in the hardware, the state of the simulation instance does not change until there is an actual change in the hardware. Following the DES paradigm, this implies that the mere act of polling might be neglected to reduce the simulation overhead.

Therefore, we propose that at the beginning of a polling phase the user code transfers control to the simulator until a change in the (simulated) hardware occurs. Such a hardware change is, analog to simulated interrupts, represented by an event in the future event set. When the event handler processes this event, the simulation time is updated and the control is handed back to the user code, which may then react depending on the hardware change. The control flow of

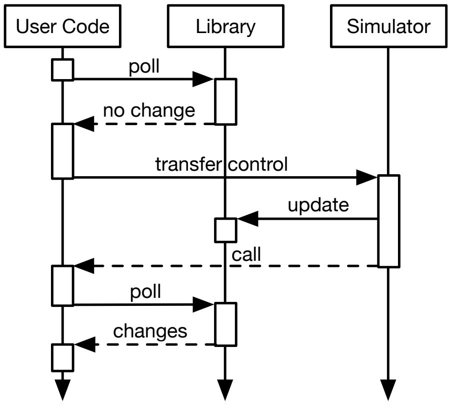
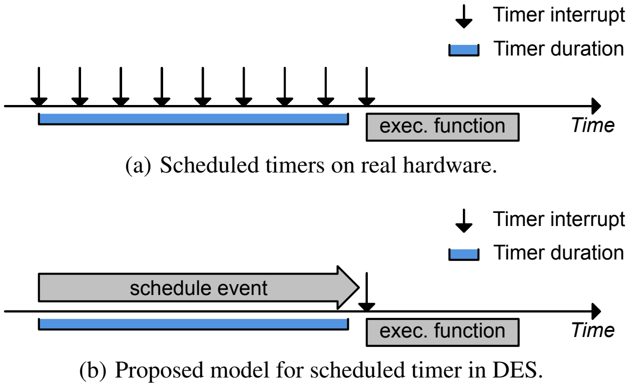


**Figure 2: Control flow of polling in the simulator. When a polling phase begins, the user code transfers control to the simulator until a hardware change occurs. Then, the control is handed back to the user code to react upon the change.**



(a) Scheduled timers on real hardware.

(b) Proposed model for scheduled timer in DES.

**Figure 3: While on real boards timers periodically fire to check whether a function is scheduled for execution (a), the proposed model for simulated timers only fires at the time of the scheduled execution (b).**

the simulated polling process is depicted in Fig. 2. It includes a user code, a library, and a simulator instance. The user code instance polls the library instance once for hardware changes, then it transfers control to the simulator instance until a hardware change occurs and the control is handed back to the user code. This allows gaining the same advantages as when simulating interrupts, while still modeling the polling process through the simulation time.

## 2.3 Timers and Schedulers

Another important component of embedded systems are timers. Timers allow the user code to schedule a certain function at a specific point in time. Moreover, timers are also used to periodically execute a certain functionality, e.g., sending a Beacon message every 100 ms. In the 802.11 Reference Design, for example, a coarse (200 ms accuracy) and a fine (64 $\mu$s accuracy) grain timer are provided. The accuracy defines the time interval, i.e., when the timer is fired to check whether a scheduled function should be executed or not. This process is conceptually illustrated in Fig. 3 (a). A direct modeling of this approach into DES would imply the creation of a new event each time the timer fires. To avoid this large and unnecessary overhead, we propose to create a single event for the scheduled time of the respective function, thus skipping void interrupts for the timers. Fig. 3 (b) depicts the proposed behavior of a timer in the simulation. Note that the simulation time only needs to be updated when the event handler processes the timer event leading to the execution of the scheduled function.

Additionally, hardware boards typically also provide a system time. The 802.11 Reference Design, for example, offers a `system time` and a `MAC time`, where the user code can adjust the `MAC time` while the `system time` is read-only. These times are realized each by a 64-bit register, which is incremented every microsecond. To simulate the progression of time in DES, it suffices to update such time registers only when they are accessed by the user code, by adding the simulated time between the previous access and the current access to the according register. This again considerably reduces the number of generated events, without constraining the simulation of the user code.

## 2.4 Memory Management

Memory is a trade-off between size, access times, and persistence. Especially in the design of embedded systems, different types of memory, e.g., Random Access Memory (RAM), Electrically Erasable Programmable Read-Only Memory (EEPROM), and flash, are used for different purposes. Depending on the simulation model, the characteristics of a specific memory type should be considered. For a code-transparent simulation, it is indispensable to map the memory address space from the embedded device to a simulated storage location.

In SDRs, mainly hardware registers and RAM are used for memory. The available EEPROM is used for contents that should persist on the boards even when the power is turned off, e.g., the board MAC address. To achieve a similar behavior in the simulation, we propose to write the content of the EEPROM to a file on the host computer in order to persist various simulation runs.

Regarding the RAM, an essential feature of the simulator is to only provide those parts of the memory that are currently needed in the simulation. To be more specific, the simulation engine shall dynamically allocate and initialize only the needed memory parts. During our analysis of the WARP library code, for example, we found that the large parts of the available RAM are initialized, although these parts of the RAM are never used for write and read operations. Thus, such memory parts do not need to be allocated and initialized in the simulator. In this way, the overall used memory of all simulated instances can be kept low, which, in turn, ensures efficient memory usage on the host system.

Similarly, typically not all hardware registers are used during the execution of the user code. Consequently, only those registers that are accessed by software or by (simulated) hardware components need to be allocated in the simulation. For code-transparency, such simulated registers can be realized in a map, where each entry consists of the register's address and the corresponding value.

## 2.5 FPGA Cores

FPGAs are increasingly integrated into SDRs, as they provide a flexible way in realizing certain tasks, e.g., signal processing, in the hardware [25]. While some of these tasks might be abstracted in the simulation, such as the radio transceivers, other functionality needs to be considered by the simulator.

As shown in Fig. 1, for example, essential parts of MAC protocol are realized in the FPGA of the 802.11 Reference Design. In particular, the Distributed Coordination Function (DCF) of IEEE 802.11-2012 [10] relies on a MAC support core in the WARP v3 FPGA. This core is essential for the correct protocol behavior of the board and must be therefore modeled in the simulation.

A typical approach for specifying a hardware design is by using Finite State Machines (FSMs) [22]. The advantage of such an approach is that it can be easily translated into an imperative programming language, such as C, and nicely fits the DES paradigm. For the code-transparent simulation of an SDR platform, the different hardware FSMs, depending on the SDR design, can thus be directly modeled into the simulator. With this approach, the user code is still able to parameterize the simulated hardware support cores, but design changes in the FPGA consequently require changes in the simulation framework.

## 2.6 Processor

The main component of an SDR platform is the processor, enabling the execution of programmable user code. For a code-transparent simulation, the processor of the SDR board needs to be represented in the simulator. The speed of a processor mainly depends on its clock rate and on its architecture, e.g., whether it supports floating point operations or not. One way to accurately model the processor behavior is by Instruction Set Simulation (ISS) [4]. However, such fine-grained models create a large overhead for the simulation engine, especially when simulating a large distributed topology. Consequently, we opt for not modeling the execution of every processor instruction, but instead to measure the execution time for time-consuming tasks, e.g., copying data from one memory location to another, and to calibrate our DES engine accordingly. Further details regarding the calibration process are provided in Sec. 4.1.

## 2.7 Hardware I/O

Finally, an SDR platform consists of various hardware I/O components, which enable the board to interact with its environment. Such components typically include, for example, General-Purpose Input/Output (GPIO) pins, a Liquid Crystal Display (LCD), Universal Asynchronous Receiver / Transmitter (UART), Ethernet ports, and an antenna. For each component, the input and output characteristics must be analyzed to allow for an adequate modeling in the simulator. Output that is only meant for logging purposes, e.g., GPIO pins, LCD, and UART, might be simply written to text files in the simulation for a subsequent evaluation. For input, e.g., a user pressing a button, the simulation might include a user model pressing the button when a certain condition applies.

# 3. DESIGN

In this section, we present our design for *Code-transparent Wireless Prototype Simulation (CoWS)*, which builds upon the analysis in Sec. 2. Although the proposed design mostly offers general approaches for the simulation of wireless prototypes, we show its feasibility by describing the integration of the existing 802.11 Reference Design for WARP v3 into the popular DES framework ns-3. In the following, we first provide a short overview of the main design components. Then, we describe each component more detailed in individual sections.

## 3.1 Overview

The main architecture of CoWS consists of three layers, which are depicted in Fig. 4, namely SDR code, DCE, and ns-3. On the SDR code layer, we further differentiate between *user code* and *library*. The user code is by design *code-transparent*, i.e., the same code runs on SDR hardware as in the simulation. In the library code, in turn, the function prototypes are unchanged while the implementations might be adapted to call the simulation engine instead of real hardware components. The clear separation between user code and library thus facilitates the code-transparent simulation of any user code that is also based on the SDR design. For the code-transparent simulation of different SDR platforms, the respective library code and hardware functions need to be analyzed (cf. Sec. 2) and similarly interfaced to the DCE layer.

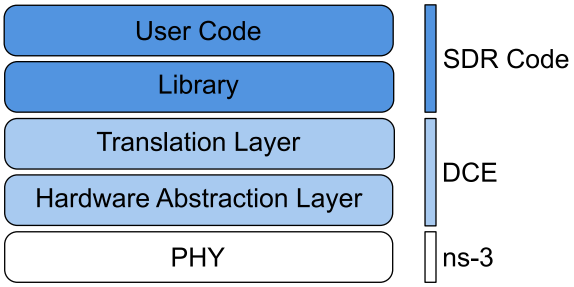


**Figure 4: Proposed architecture of CoWS. The design consists of three main layers: ns-3, DCE, and the SDR Code.**

The DCE layer includes an adaptation of the ns-3 extension DCE by Lacage [14] to enable the code-transparent simulation. It is structured into a *translation* and a *hardware abstraction layer*. The translation layer maps function calls from the SDR code layer to their counterpart in the simulation, while the hardware abstraction layer models the behavior of hardware components using DES.

Finally, the ns-3 layer provides the simulation engine, which is used to model all network-related functions, e.g., the physical topology of the network or the respective transmission channels. As the modifications on this layer are only marginal, it may be easily replaced by another network simulation engine, e.g., OMNeT++ [1], given that a similar extension like DCE is available to the simulator. A more detailed description of the three layers is provided in the remainder of this section.

## 3.2 SDR Code Layer

To integrate the SDR source code into ns-3, the code needs to be analyzed in order to identify those code fragments that access different hardware components. In general, the user code accesses such components through a library that implements different functions. In the WARP v3 design, this library is divided into a library for CPU High, and a library for CPU Low, as the design implements two CPUs. Apart from such hardware accesses, there are other factors in the code execution that prevent the user code to be directly simulated with DES. In the following, we shortly describe different aspects of the user and library code that need to be addressed in code-transparent DES.

### *C Libraries.*

The user code as well as the library code might both use C libraries, most prominently the C standard library. As such libraries are typically not specific to certain hardware, they can be directly integrated into the simulation. In the 802.11 Reference Design, for example, the user code frequently accesses the C library function `malloc` to allocate memory. In the simulation, such functions are typically also available on the host system. However, some library functions need to be modified to match the DES paradigm. Yet, to simulate an SDR platform in ns-3, there is typically no need to adapt the libraries, as DCE already provides the necessary adaptions.

In this context, it is important to note that the system architecture of the host system might be different from the simulated SDR platform. This might have an effect on the sizes of certain data types, e.g., the SDR platform might have a 32-bit architecture, while modern computer systems typically provide a 64-bit architecture. To avoid unexpected behavior when switching from embedded device to simulation, data types should have a specified size, e.g., by using `int32_t` instead of `int`.

### *Polling in Infinite Loops.*

To simulate the user code with a DES paradigm, a major problem occurs when the code is indefinitely polling for hardware changes, which we already described in Sec. 2.2. According to the DCE manual [7], infinite loops need to be modified in a way that the user code periodically hands over control to the simulator engine in order to let the event handler process the next event. Otherwise, a deadlock situation occurs where the user code polls for hardware changes, which can not happen because future events are not processed as long as the current event, i.e., indefinite polling, does not complete.

There are several ways in how to hand over control to the simulation engine. One possibility is let each polling function in the library code call the simulation engine. This would also preserve the code-transparency of the user code. However, this might result in many unnecessary simulation engine calls and eventually slow down the simulation execution time. Therefore, we opt for slightly modifying the user code in a way that each infinite loop, e.g., the polling loop in the `main` function, calls the simulation engine after one iteration, which typically consists of polling different hardware components. The control is then only handed over to the user code when the next event is processed. As such loops are intended by the developer to poll indefinitely for hardware changes, they can be easily identified manually. Moreover, there are also ways in automatically detecting such loops, e.g., with static code analysis [3].

### *Random Numbers.*

Randomness is an important aspect of real-world communication protocols, which is difficult to adequately address in computer simulation. Indeed, in many cases it is even useful that real-world stochastic processes are controllable in the simulation, as this enables reproducibility of simulation results.

In CoWS, the user code may get random numbers, which are internally provided by a pseudo-random number generator. For most specifications, e.g., the IEEE 802.11 standard [10], such a pseudo-random number generator suffices. Additionally, this allows to set a specific seed and thereby to reproduce simulation runs. Otherwise, each simulated SDR node is seeded with another random number to ensure a different behavior when relying on the pseudo-random number generator.

## 3.3 DCE Layer

The main purpose of DCE is to enable the simulation of existing user and kernel space implementations in ns-3 [14]. Thereby, a core feature of DCE is to map Portable Operating System Interface (POSIX) API calls to ns-3. This enables the code-transparent simulation of Linux applications, e.g., iperf, in ns-3. For CoWS, we make use of DCE to enable code-transparent simulation of SDR user code. In order to do so, DCE needs to support the functionality of the respective SDR library and, where necessary, provide abstractions from the actual hardware.

Following the DCE architecture, POSIX functions are provided through shared libraries. To simulate WARP nodes, we extend DCE with an additional shared library, which we name `libwarp`. In general, DCE allows to add a system call either by using the implementation provided by the host operating system, a so-called `NATIVE` symbol, or by replacing the implementation in the library with an own implementation, a so-called `DCE` symbol [7]. The latter is applied for symbols where the behavior needs to be adapted for DES. The `libwarp` thus acts as an interface between modified WARP library and DCE layer to hand over control to the simulator. All `libwarp` symbols are of the kind `DCE`, as they do not exist in the host operating system. Their respective implementations are provided on the DCE layer. For each SDR design, a corresponding shared library needs to be implemented in DCE in order to enable the simulation of the respective SDR platform. In the following, we describe how the library and hardware abstraction of the most important functions is realized.

### *Processors and Shared Memory.*

SDR platforms might include more than one CPU to parallelize the execution of tasks. As described in Sec. 2.1, the 802.11 Reference Design, for example, implements two Xilinx MicroBlaze CPUs, namely CPU High and CPU Low, where the latter is responsible for time-critical tasks such as sending acknowledgement packets and the former is responsible for high-level management tasks such as the association of new stations. The communication between the two CPUs is enabled through shared memory, e.g., the CPUs may both access the transmit and the receive buffers, and via message passing. For the latter, the 802.11 Reference Design implements a `Mailbox`, which manages the messages for the respective CPUs in first in, first out (FIFO) queues. To receive a new message, CPU High may register for an interrupt or use polling, while CPU Low exclusively uses polling.

In CoWS, the simulation of multiple CPUs is supported by letting the code for each CPU run each in its own thread, which CoWS manages on its DCE layer. The DCE layer also centrally manages the communication via shared memory and message passing, e.g., the FIFO queue of the `Mailbox` can be easily achieved with `std::queue`. Moreover, the message interrupts and polling mechanisms nicely fit into DES, when modeled as described in Sec. 2.2.

### *Radio Communication.*

Most parts of the radio communication are directly handled by ns-3, and can thus be configured as one would usually configure such communication in ns-3. However, when handing over a packet from the user code to ns-3, and vice versa, translation steps are necessary. We address these changes on the DCE layer, which acts as a translation layer between the user code and ns-3.

Moreover, the simulation engine, similar to a real SDR board, needs certain information to be able to transmit a radio packet. This metadata, e.g., the total packet length, the employed Modulation and Coding Scheme (MCS), the transmission power, etc., is thus used to configure the transceiver. On the DCE layer, such parameters need to be passed from the user code to the simulated PHY.

Similarly, for each received packet the PHY captures certain metadata, e.g., the measured Received Signal Strength Indicator (RSSI) during the reception. In the simulation, such information needs to be conveyed from the simulated PHY to the simulated SDR user code, besides the actual data packet. The 802.11 Reference Design defines two C structs, i.e., `tx_frame_info` and `rx_frame_info`, which are stored in the transmit and accordingly in the receive buffer along with the data packet for information exchange between PHY and user code.

### *FPGA Support Cores.*

Given that the SDR design includes FPGA support cores for the user code, these support cores need also to be considered for code-transparent simulation, as already mentioned in Sec. 2.5. The FPGA of the 802.11 Reference Design, for example, includes a MAC Support Core, which handles sending of radio packets according to the IEEE 802.11-2012 standard [10]. More specifically, it implements low-level MAC mechanisms of DCF, i.e., the access to the shared medium according to the protocol. The implementation into the

FPGA fabric guarantees a deterministic timing, which is essential for respecting the Inter-Frame Spaces (IFSs). The MAC Support Core is composed of three controllers, where each consists of an FSM handling the transmission of different packet types.

The simulation of an FPGA configuration, which is specified in Hardware Description Language (HDL) or Verilog, is not supported in a code-transparent way in CoWS. Remember that the code-transparent simulation of an FPGA is not the focus of this work, but rather the code-transparent simulation of the user code. FPGA support cores are therefore considered as another (static) hardware component that needs to be modeled in the simulator. A convenient way to do so is to use an automatic tool that converts HDL or Verilog into C/C++. Based on the translated code, we implement the support cores in the DCE layer. Each simulated SDR node then relies on an own instance of the simulated cores.

*Ethernet Communication.*

SDR platforms often provide one or more Ethernet ports, which are primarily used to transparently forward Ethernet packets via the radio and to exchange configuration / measurement data with a desktop computer. In the 802.11 Reference Design (cf. Fig. 1), CPU High is responsible for receiving and sending packets via the Ethernet ports. For receiving, interrupts may be enabled through the user code, which CoWS realizes as described in Sec. 2.2.

As ns-3 supports the simulation of Ethernet hosts using the Tap Bridge Model, this part can be reused for CoWS. Again ns-3 separates the Ethernet header from the payload, which requires a small translation step from standard-compliant Ethernet packets (as expected by the user code) to the ns-3 representation of Ethernet packets. Apart from this minor modification, Ethernet communication with simulated SDR nodes seamlessly integrates into ns-3, as further explained in Sec. 3.4.

## 3.4 ns-3 Layer

The ns-3 layer contains the DES core and is further responsible for the simulation of the communication network. This includes the different network components, e.g., an SDR instance, a physical topology of the simulated network, and the respective channel models. As mentioned before, ns-3 already provides flexible configurations for these components, which we can directly integrate into CoWS. It is, for example, possible to include simulated SDR and ns-3 Wifi nodes in the same topology. In the following, we describe the integration of SDR nodes into ns-3 network topologies and furthermore provide details on the simulated PHY and possible channel model configurations.

*Simulation of a Network Component.*

In ns-3, simulated nodes use the `NetDevice` class for communication with other nodes. Therefore, an ns-3 node includes for each communication channel an instance of `NetDevice`. For standard Wifi communication, for example, a node may include an instance of `WifiNetDevice`, which is derived from `NetDevice`. When simulating a specific SDR design, it might be necessary to provide a custom derivation of `NetDevice` for the respective radio interface.

The architecture of a simulated WARP node in CoWS is depicted in Fig. 5. It consists of the user code, the WARP library code, DCE hardware abstractions, and instances of `NetDevice` for communication to other nodes. For the radio communication, we created the `WarpNetDevice` class, which is similar to `WifiNetDevice` but uses the MAC implementation and PHY configuration defined in the WARP user code and on the DCE layer. For the 802.11 Reference Design this corresponds to the IEEE 802.11-2012 standard [10], but as the user code and the FPGA can be modified, this might follow any custom-made MAC protocol and PHY configuration. Moreover, simulated WARP nodes include a `CsmaNetDevice`, an ns-3 module for an Ethernet connection to other nodes.

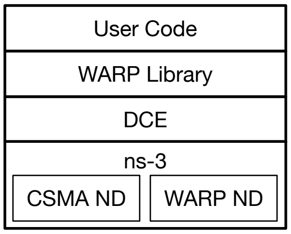


**Figure 5: Architecture of a simulated WARP node in CoWS.**

*PHY and Channel Models.*

To model a wireless channel in ns-3, the user may either implement its own channel model or rely on already existing ones. When considering IEEE 802.11, ns-3 provides `YansWifiPhy` for the corresponding simulation of the PHY and `YansWifiChannel` for the simulation of a wireless propagation model [15]. These classes offer a wide range of configuration options, which allow representing a similar setup in the simulation to the setup of the real prototyping boards. In the next section, we provide more details on the configuration of `YansWifiPhy` and `YansWifiChannel`, which we use for the validation of CoWS.

# 4. VALIDATION

The aim of this section is to validate CoWS in real-world deployments, i.e., we use different performance metrics to assess the validity of the results obtained by CoWS in comparison to the real world. Before describing our validation results in more detail, we first discuss the applied validation methodology.

## 4.1 Methodology

As explained in Sec. 3, major parts of the hardware design functionalities are modeled in CoWS to enable code-transparent simulation of the user code. Regarding their timings, we differentiate between four distinct cases: (i) *inherent timings*, e.g., a hardware timer of 100 $\mu$s, (ii) *standard-defined timings*, e.g., a DCF Inter-Frame Space (DIFS) in IEEE 802.11, (iii) *nature-defined timings*, e.g., wireless propagation speed, and (iv) *hardware-dependent timings*, e.g., a channel switch on the transceiver. While (i) to (iii) can be directly configured in the DES engine, (iv) requires measurements on the target platform to calibrate the simulation accordingly. Such measurements strongly depend on the selected hardware and therefore need to be performed for each platform prior to its accurate simulation. As an example, we provide details on the calibration of the 802.11 Reference Design (v1.5.2) on WARP v3 hardware, followed by the considered scenarios and metrics. Note that it is also possible to calibrate simulation code that is purely based on ns-3. However, we will show that with calibration, our code-transparent approach performs at least as well as a pure ns-3 solution.

*Calibration.*

For the calibration of CoWS, we first analyze which WARP design functionalities might have an impact on the execution time. Where necessary, we measure the actual execution time of the selected functionality in the real hardware. We repeat each measurement 1000 times to compute an average value; the variance is for all measurements negligibly small. Table 1 summarizes our findings.

| Design component | Timing [$\mu s$] |
|---|---|
| Read/write mailbox | $\approx 1$ |
| Channel switch | 93.6 |
| Radio transmissions | time modeled by ns-3 |
| Ethernet transmissions | time modeled by ns-3 |
| Print text via UART | $86.999x + 0.013$ |
| Direct Memory Access | $1.240; \quad x < 1\,\mathrm{MB}$ |
| | $2.862; \quad x \geq 1\,\mathrm{MB}$ |
| memcpy | $0.007x + 0.012; \quad x < 100\,\mathrm{Byte}$ |
| | $0.05x - 0.073; \quad x \geq 100\,\mathrm{Byte}$ |

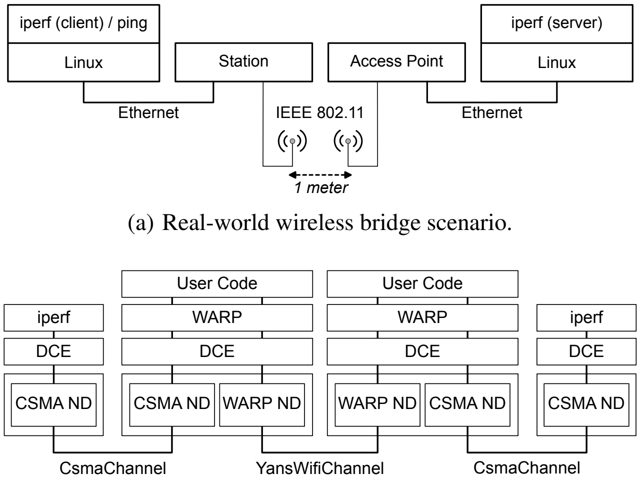
**Table 1: Measured timings in $\mu s$ of different 802.11 Reference Design components on WARP v3, where $x$ denotes the number of Bytes used in the respective operation.**

Note that the timing of some functionalities also depends on the size of the input, e.g., printing text via UART depends on the number of Bytes $x$. The timings are then used to calibrate the simulation time of respective events in ns-3. However, not all design aspects that require execution time are considered in Table 1. It is known, for example, that floating point operations need a relatively long computation time on WARP nodes due to the missing hardware support for floating point operations. Nevertheless, this issue is not considered in CoWS, as our DES model abstracts from the CPU instruction level as already explained in Sec. 2.6.

### *Validation Scenario.*

We base our validation on an existing real-world installation, namely the *wireless bridge* scenario [1], which is depicted in Fig. 6(a). In this scenario, two WARP nodes are each connected to a computer via Ethernet. This allows to run any application on the computers, e.g., iperf, to test or benchmark the radio communication between the WARP nodes. Therefore, Ethernet packets from one computer to a WARP node are transparently forwarded via the radio to the other WARP node. This other board then forwards the packets via Ethernet to the application on its connected computer. From the application layer perspective, the replacement of the Ethernet connection by this wireless WARP bridge does not require any changes in the application code, as the 802.11 Reference Design includes encapsulation and decapsulation functions to covert Ethernet to WiFi packets and vice versa. To compare the results obtained in the real-world scenario with a simulated scenario, we additionally simulate this scenario in CoWS. The architecture of the simulated nodes and their topology is depicted in Fig. 6(b).

Both scenarios of Fig. 6 consist of an AP and a STA. The STA is wirelessly associated to the AP at a distance of 1 m. We selected these idealized conditions, as the aim of this paper is not to model a specific propagation environment in ns-3, but to validate the behavior of the simulator in a scenario comparable to the real-world scenario. We perform our measurements in the 2.4 GHz and in the 5 GHz band to validate our approach under various influences on the wireless propagation. The Operating System (OS) on each computer is an Ubuntu Linux with kernel version 2.6.35, as this is the closest match to the default Linux kernel version of DCE kernel mode (version 2.6.36) [7]. In total, we differentiate between four configurations of the mentioned scenario:



(a) Real-world wireless bridge scenario.

(b) Simulated wireless birdge scenario.

**Figure 6: Validation scenario: A STA is wirelessly connected to an AP. For performance measurements, e.g., with iperf or ping, both devices are connected to a computer via Ethernet.**

**WARP Antenna.** The real-world wireless bridge scenario, where the WARP nodes use their antennas for Single Input and Single Output (SISO) communication. The antennas are placed at a distance of 1 m to each other. Additionally, each WARP node is connected via Ethernet to a computer with Linux kernel version 2.6.35.

**WARP Cable.** This configuration equals to the WARP Antenna configuration, except that the antennas are replaced by a coaxial cable to shield the radio communication from external interference. Moreover, an attenuator of 42 dB is included in the cable connection to avoid damage to the radio controllers.

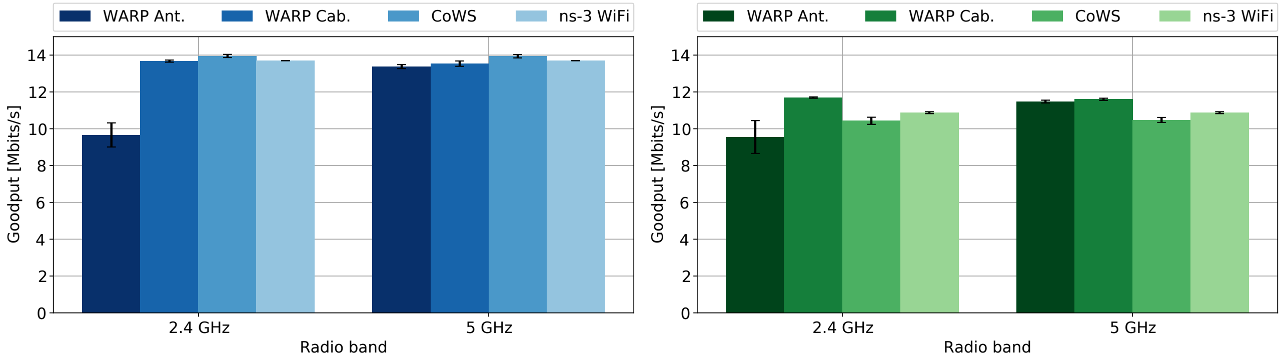
**CoWS.** A simulation of the wireless bridge scenario in CoWS, as shown in Fig. 6(b). As wireless channel model, we use `YansWifiChannel` with log distance propagation model and constant speed propagation delay for communication at a simulated distance of 1 m. Each WARP node is connected to a simulated node via an ns-3 `CsmaChannel`. The simulated hosts run with kernel version 2.6.36.

**ns-3 WiFi.** This configuration equals to the CoWS configuration, with the difference that the simulated WARP nodes are replaced by ns-3 `WifiNetDevices`. Consequently, they do not run the 802.11 Reference Design, but they are, however, also simulating IEEE 802.11 and configured to use the same MCS, transmit power and wireless channel as the WARP nodes. We use this configuration to investigate how close the provided ns-3 models match real-world scenarios.

### *Metrics.*

Based on the previously described calibration and validation scenarios, we are mainly interested in two performance metrics to validate the behavior of CoWS compared to real-world deployments. The first one is the achieved *throughput*, or more specifically, the *goodput* we achieve in real-world configurations compared to simulations. Secondly, we are interested in how well the latency is modeled in CoWS compared to the real-world and to the ns-3 WiFi simulation. In the four validation configurations, we measure goodput and latency using the tools iperf and ping, respectively. In the following, we present the measured results of the tools, each in an individual section.

[1] https://warpproject.org/trac/wiki/OFDMReferenceDesign/Applications/Bridge



(a) UDP measurements.

(b) TCP measurements.

**Figure 7: Mean UDP / TCP goodput for each of the four scenario configurations in the 2.4 GHz and the 5 GHz band.**

## 4.2 Goodput

For each of the four configurations that we described in the previous section, we measure the achieved *goodput*, i.e., the rate of successful data delivery on the application layer. The measurements are performed with iperf, where the iperf server runs on the computer / simulated host that is connected to the AP. The iperf client, in turn, runs on the computer / simulated host that is connected to the STA. Following the default configuration of the 802.11 Reference Design, we set the MCS to Quadrature Phase Shift Keying (QPSK) and with $3/4$ coding, which achieves a theoretical PHY throughput of 18 Mbit/s. We repeat the measurements for a single configuration 30 times to compute an average and a standard deviation; the goodput results are extracted from the iperf server report.

### *UDP Measurements.*

The results of the User Datagram Protocol (UDP) goodput are shown in Fig. 7(a). When comparing the WARP Antenna results for the 2.4 GHz and 5 GHz band, we note that the achieved goodput is about 3 Mbit/s lower than in the 5 GHz band. We attribute this observation to the increased interference in the 2.4 GHz band due to co-existing radios. The results in the 5 GHz band, which is much less crowded by other technologies, are almost the same as for WARP Cable, where no interference occurs. As interference is currently not considered in the channel model of our simulation, the channel of the WARP Cable configuration actually closely matches the channel model in CoWS and ns-3 WiFi. Still, we observe a slightly higher goodput with CoWS compared to WARP Cable, on average about 0.28 Mbit/s higher with 2.4 GHz, and on average about 0.4 Mbit/s higher with 5 GHz, as the additional attenuation of 42 dB in the cable setup is not modeled in the simulation, which results in slightly better propagation conditions in the simulation.

In comparison, the goodput of the ns-3 WiFi configuration is on average 0.25 Mbit/s lower than with CoWS, as we are considering two independent implementations of IEEE 802.11 with slight differences. These results show that CoWS achieves a close match in the UDP goodput compared to the real world, simulating the calibrated 802.11 Reference Design with a deviation below 3 %.

### *TCP Measurements.*

Compared to UDP, which is a stateless transport protocol with low overhead, the main features of Transmission Control Protocol (TCP) are retransmissions, and flow / congestion control to provide reliable, in-order data streams between a client and a server. However, these mechanisms might come at the price of a reduced goodput compared UDP, as observed in the results of Fig. 7(b). There, in the WARP Antenna configuration, the goodput is at 9.55 Mbit/s in the 2.4 GHz band, and at 11.48 Mbit/s in the 5 GHz band. Even the WARP Cable configuration only achieves about 11.65 Mbit/s in both bands. In comparison to WARP Cable, the results of CoWS and ns-3 are lower with about 10.45 Mbit/s and 10.87 Mbit/s, respectively. Tazaki et al. also observed in [23] that simulating TCP in DCE kernel mode yields a lower goodput than the corresponding real-world setup, although both use the same kernel version. This indicates that the DCE kernel mode needs to be further revised to obtain better-aligned goodput results.

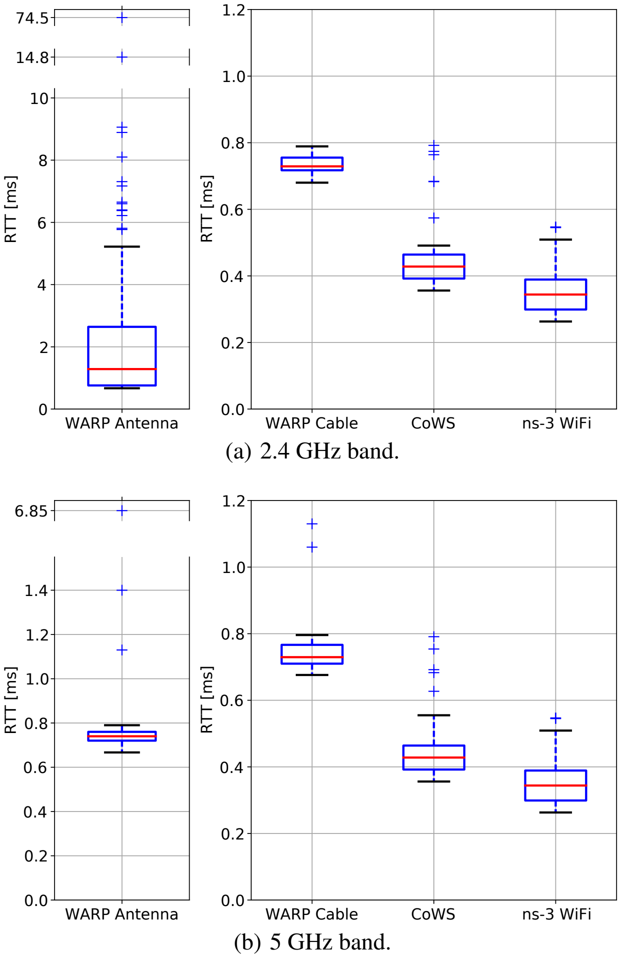
In conclusion, the results show that CoWS achieves comparable goodput results to a real-world scenario, although we detected slight differences due to the model abstractions. The ns-3 WiFi approach also achieves comparable results to the WARP Cable configuration, but it lacks code-transparency whereas in CoWS we can support any protocol that was developed for the hardware platform. Nevertheless, the communication latency needs to be further investigated, especially regarding the simulation of the Linux communication stack, which we do in the following section.

## 4.3 Latency

For the communication latency measurements, or more precisely the RTT, we consider again the four configurations presented in Sec. 4.1. In these configurations, we measure the RTT between the STA and the AP using ping. Apart from this modification, the four configurations remain unchanged. We repeat the measurements for each configuration in the 2.4 GHz and in the 5 GHz band 100 times. The results are plotted in Fig. 8(a) and Fig. 8(b), respectively.

The WARP Antenna results in Fig. 8(a) and Fig. 8(b) once again show the effects of a crowded transmission band, i.e., at 2.4 GHz, in combination with a random back-off scheme on the MAC layer. The average latency of the WARP Antenna configuration in the 5 GHz band, which is less affected by wireless interference, is almost identical to the latency of the WARP Cable configuration, i.e., about 750 $\mu$s. However, in comparison, the latency for CoWS is significantly lower at about 430 $\mu$s and the latency for ns-3 WiFi reaches an average of 340 $\mu$s. We attribute this time difference to the fact that CoWS models more exactly different (hardware) design functionalities due to the calibration process. Nevertheless, there remains a time difference of about 320 $\mu$s between CoWS and WARP Cable, which we further investigate in the following.

We suspect that in DCE kernel model, the processing time in the Linux kernel is not accurately modeled in ns-3. To confirm this hypothesis, we conduct an additional experiment that focuses



(a) 2.4 GHz band.

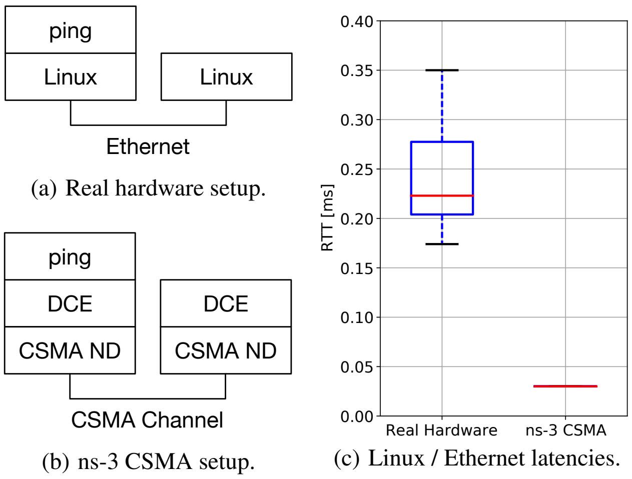
(b) 5 GHz band.

**Figure 8: RTTs of the four configurations in the 2.4 GHz and in the 5 GHz band. The measurements are based on using ping.**

on the processing times in the Linux communication stack. The measurement setup is shown in Fig. 9(a). It consists of two hosts that are directly connected via Ethernet. We implement this setup with real hardware and, for comparison, with DCE kernel mode, again with Linux kernel version 2.6.35 for both setups. The measurement setup in the simulation is depicted in Fig. 9(b). We measure the RTT between the two hosts with ping, where we repeat the measurements 100 times for both the real and the simulated setup. The obtained results are shown in Fig. 9(c).

In the real hardware setup, the average RTT is 237 $\mu$s whereas, in the ns-3 setup, we measure a constant RTT of 30 $\mu$s. The time difference between the two setting is thus about 207 $\mu$s. We attribute this time difference mainly to processing within the Linux communication stack and on the network cards, as the propagation time over a 1 m twisted pair cable is below 10 ns and can thus be neglected. This indicates that such processing times are not considered in ns-3, thus leading to shorter RTTs.

When we thus add the processing time difference of 207 $\mu$s to the measured RTT of CoWS (cf. Fig. 8), there still remains a time difference of 100 $\mu$s to the average RTT of the WARP Cable configuration. We account this difference to the CPU processing on the WARP nodes, which is limited to a certain speed on real world hardware, but is, however, not modeled in CoWS, as we abstract from CPU instruction level in the DES.



(a) Real hardware setup.

(b) ns-3 CSMA setup.

(c) Linux / Ethernet latencies.

**Figure 9: RTTs between two Ethernet-connected computers in real world and in ns-3 (c). The real hardware and the ns-3 CSMA setups are shown in (a) and (b), respectively.**

In total, the latency measurements show that with CoWS we can accomplish a similar timing behavior as in a hardware setup, with a corrected deviation of about 13 %. To achieve a closer match between simulation and real hardware timings, the involved setup components, e.g., the Linux network stack, need to be carefully analyzed and calibrated in ns-3.

# 5. EVALUATION

After the validation of CoWS with real-world setups in the previous section, we are further interested in the efficiency of our implementation. Remember that a central benefit of network simulation is the ability to flexibly scale the simulation scenario to a required (large) network size. Especially when working with prototyping platforms, such as WARP, medium-size networks, i.e., already with more than ten nodes, are expensive and difficult to maintain. Therefore, an important performance indicator for CoWS is the scalability, i.e., how does our code-transparent approach affect memory consumption and computation time depending on the network size? In the following, we first describe the chosen evaluation setup. Then, we discuss the measured results for the memory consumption and the computation time.

## 5.1 Setup

The evaluation setup consists of a simulated $5\,\mathrm{m} \times 5\,\mathrm{m}$ area in which $n$ simulated nodes are randomly placed. We consider various configurations of the network size, in the range of $1 \leq n \leq 1000$. In each configuration, we have one AP and the remaining nodes are STAs. The AP and STA user codes are simulated with CoWS as described in Sec. 3, based on version 1.5.2 of the 802.11 Reference Design, DCE version 1.8, and ns-3.25. For comparison, we simulate this scenario also through ns-3 WiFi, using `WifiNetDevices` for AP and STAs. The simulation runs on a Ubuntu Desktop 14.04 machine with an Intel i7 3.1 GHz CPU and 16 GB RAM. We repeat the measurements for each configuration 30 times.

## 5.2 Memory Usage

We measure the peak memory consumption of the ns-3 process that is responsible for simulating the WARP nodes. In the considered time frame, CoWS has initialized the needed data structures for each simulated WARP node and the user code is in its `main` function

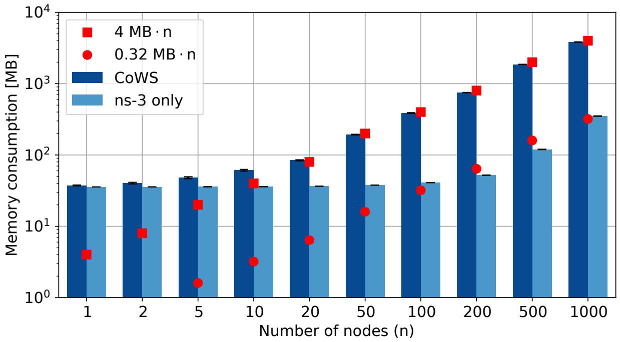


**Figure 10: Total memory consumption for simulated networks of different sizes.**

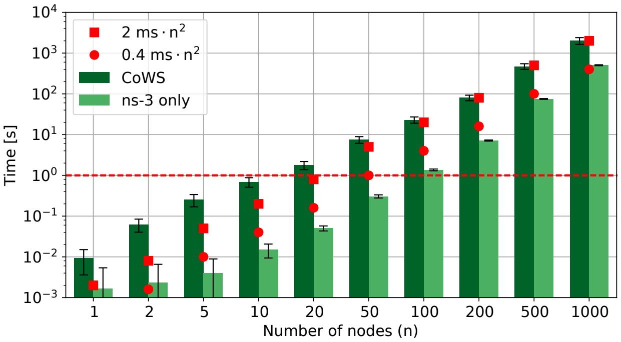


**Figure 11: Total computation time for the simulation of 1 s (dashed line) in networks of different sizes.**

polling for inputs or accordingly waiting for interrupts. Similarly, we also measure the peak memory consumption of the ns-3 WiFi scenario. The results for both simulation scenarios, depending on the number of nodes, are depicted in Fig. 10.

In both scenarios, we see that for small network sizes (up to 10 nodes), the memory consumption is dominated by a constant amount of required memory for the ns-3 process. Then, a linear growth of the occupied memory with each additional node can be observed. For CoWS, it can be approximated by the linear function

$$f(n) \quad = \quad 4\,\text{MB} \cdot n \quad , \tag{1}$$

where $n$ denotes the number of simulated nodes. Hence, each simulated WARP node occupies about 4 MB of memory, although the real WARP v3 boards have a RAM of 2 GB at their disposal. However, as described in Sec. 2, CoWS only allocates the amount of memory that a simulated WARP node currently needs. Thus, we are able to reduce the memory allocation per WARP node to about 4 MB. In the ns-3 WiFi scenario, we observe that each simulated node requires roughly 300 kB memory, as its growth depending on $n$ can be approximated by

$$g(n) \quad = \quad 0.32\,\text{MB} \cdot n \quad . \tag{2}$$

These results show that, as expected, both scenarios have a linear growth in their memory consumption, where for code-transparent simulation the amount of needed memory per board can be kept low when analyzing the library code accordingly (cf. Sec. 2.4).

## 5.3 Computation Time

Besides the memory usage, an important factor for the scalability of our approach is the required computation time, as, in DES, the simulation time does not necessarily match the actual time to execute the simulation, which is influenced by many factors. Thus, we measure the computation time for a simulation of 1 s in networks of different sizes. During this time, each simulated node periodically broadcasts messages to the other nodes, as all nodes are in wireless reception range of each other. The obtained results for CoWS and ns-3 WiFi are shown in Fig. 11.

For larger network sizes (above 20 nodes), the computation time for both scenarios grows quadratically with $n$. At this topology size, the simulation time corresponds to the computation time, which, however, strongly depends on the underlying scenario, as further discussed in the remainder of this section. The computation time of CoWS can then be approximated by the following function:

$$f(n) \quad = \quad 2\,\text{ms} \cdot n^2 \quad . \tag{3}$$

For configurations below 20 nodes, the computation time is relatively higher, as other ns-3 tasks, which are independent of $n$, dominate the measurements. In the ns-3 WiFi scenario we also have a quadratic growth, with a lower constant factor than in CoWS, which can be roughly approximated to

$$g(n) \quad = \quad 0.4\,\text{ms} \cdot n^2 \quad , \tag{4}$$

as in simulation approaches that are not code-transparent, typically less DES events are required due to the higher abstraction level.

The quadratic growth of the computation time in both scenarios is caused by the communication complexity: As each of the $n$ nodes broadcasts messages to $n - 1$ nodes, the total communication complexity is $\mathcal{O}(n^2)$, where we need at least one event per message. These results show that even computationally expensive scenarios with a quadratic communication complexity can be simulated in CoWS in an acceptable time on commodity hardware for typical wireless network sizes, i.e., below 100 nodes. For larger networks, however, more advanced simulation techniques should be considered in CoWS as well as in ns-3 WiFi, e.g., parallel DES [19].

# 6. DEBUG FUNCTIONALITY

Besides scalability, a further advantage of simulation compared to prototypical deployments is the enhanced ability to find implementation defects. For distributed systems, it is difficult to concurrently inspect the state of interacting entities, while simulation of such systems, in turn, offers a global view on the current state of the network due to implicit synchronization, which significantly facilitates the analysis. Furthermore, when pursuing a *code-transparent* approach as in CoWS, the simulation may also be used to find *implementation* defects to improve the common code basis for simulation and prototypes. On the host computer, which is used for the simulation, already advanced debugging tools are provided through the OS, while direct debugging of embedded systems is typically a rather cumbersome process. In the following, we describe our preliminary results of searching for implementation defects with CoWS.

As an example to test the debug functionality enabled by CoWS, we analyze the 802.11 Reference Design with GNU Project Debugger (GDB) [8]. Therefore, we configure a simple scenario consisting of a STA and an AP, which we run with CoWS on a Ubuntu Desktop 14.04 machine with an Intel i7 3.1 GHz CPU and 16 GB RAM. During the execution of the simulation, we attach GDB to the STA and to the AP to inspect the program state during run-time. We found several implementation defects in the user code as well as in the WARP library. In general, each detected defect should be analyzed regarding its impact on the hardware board and reproduced on the board to rule out defects introduced by the simulation.

These defects were mainly caused by access to uninitialized pointers or by dereferenciation of `null` pointers. For embedded devices, in contrast to systems that are managed by an OS, it is typically possible to write to any addressable part of the memory. However, writing to arbitrary parts of the memory may lead to unexpected and erroneous behavior of the device, which is, in addition, difficult to track down. In particular, we found, in the 802.11 Reference Design code (version 1.5.2), dereferencing of an uninitialized pointer[2], wrong usage of a memory comparison function[3], and an erroneous register access[4]. All reported code defects were acknowledged and fixed by the WARP project in code version 1.5.3[5].

Although the focus of this work was not to thoroughly test a specific protocol implementation, our small case study already shows the potential of CoWS. We expect that the integration of more systematic testing approaches, e.g., symbolic execution [12], will enable CoWS to exhaustively test code of prototyping hardware regarding implementation defects and ultimately improve the reliability of such deployments.

## 7. RELATED WORK

CoWS enables code-transparent simulation of wireless prototypes to better analyze and to flexibly scale existing hardware deployments. Therefore, we compare our approach to advanced analysis tools for wireless prototype networks as well as code-transparent simulation techniques in this context, which we present in the following.

The Power Aware Wireless Sensors (PAWiS) simulation framework [28] allows simulating various aspects of Wireless Sensor Networks (WSNs) based on OMNeT++. The focus of this simulation framework is to optimize WSN deployments regarding power efficiency and reliability. Therefore, a model of each WSN node, including its function, timing, and power consumption needs to specified, where the detail granularity of the model can be adapted depending on the simulation aspect that the user is interested in. Moreover, data post processing tools allow to analyze the output of a simulation run, and subsequently to optimize the WSN based on the results. Nonetheless, the specification of an adequate simulation model is a cumbersome and error-prone process, which we avoid by our code-transparent approach.

In a similar regard, Dustminer [11] provides a debug tool that analyzes logs from deployed nodes to discover events that lead to errors in the interaction patterns. The output logs need to be fed to Dustminer, which allows a post factum search for irregular behavior patterns. With CoWS, however, we envision that such analysis tools find implementation defects during run-time due to the easy access of node logs in simulation. For validation, found implementation defects in simulation may be then recreated in a real-world setup using the same code basis.

The authors of [9] propose *Wireless MAC Processor*, a framework that abstracts from a specific Network Interface Card (NIC) by providing a set of elementary actions and signals, which programmers use to define FSMs for the MAC protocol behavior. Such programs may then be ported to different NIC with no additional programming effort and even during run-time. Although not directly envisioned by the authors of the original paper, this also enables the simulation of such MAC models and thus facilitates the transition from real hardware to simulation.

Hybrid Simulation framework (HySim) [13] allows simulating the execution of code on microprocessors on instruction-level, which is referred to as ISS. This allows for a very precise simulation but also significantly slows down the simulation process, especially in distributed settings with a large number of nodes. The authors of [18], in turn, tackle this trade-off between scalability and accuracy with a cross-level simulator for the Contiki OS (COOJA), which allows for simultaneous simulation at different abstraction levels. COOJA enables simulation of a small subset of the network at a very high precision, i.e., at OS level or even ISS, while the remaining part of the network is simulated at the application level. This constitutes an interesting feature, which could be considered for the further development of CoWS.

A TinyOS Simulator (TOSSIM) was introduced in [16]. It enables a code-transparent simulation of TinyOS applications in WSNs with a DES. Therefore, it replaces low-level TinyOS components with TOSSIM components to translate hardware interrupts into simulation events. Similarly to CoWS, the authors were able to discover several bugs in TinyOS with the help of simulation. The main difference to our approach is the lack of an accurate timing behavior and that TOSSIM uses own, simplified channel models instead of a state-of-the-art network simulator.

In [21], the authors propose *WARPsim*, which, similar to CoWS, also provides a code-transparent simulation of WARP. Therefore, WARPsim offers a subset of the OFDM Ref. Design for WARP v1/2, which enables an execution of WARP code on standard desktop computers. The execution of WARP instances in different processes, which are coordinated via Inter-Process Communication (IPC), allows the simulation of small networks. However, this approach does not scale well for larger typologies as each additional WARP node requires its own process. Moreover, hardware timings are not considered in WARPsim, which leads to a limited comparability with real-world setups. Lastly, a major drawback of WARPsim is that is does not rely on an advanced network simulator, such as ns-3 or OMNeT++, which strongly limits its application.

Finally, a simulation framework for multicore systems, named Manifold, is proposed in [27]. This framework enables a full system simulation of a multicore architecture by flexibly integrating user-defined component models. The simulation takes place at CPU instruction-level and may be conducted in a parallel or in a sequential way. While Manifold offers a precise simulation of complex multicore architectures, it is unclear whether this approach is suited for simulating large network topologies due to the fine grain simulation steps that are involved. However, with CoWS we follow a similar modular approach to facilitate the adaptation and extension of our simulation framework.

## 8. CONCLUSION

In this paper, we propose a code-transparent simulation framework of wireless prototypes. We first analyze the needed steps to achieve code-transparent simulation and then present our design, which is embedded into ns-3 / DCE and enables the simulation of SDR nodes. The validation of our approach is based on WARP and real-world scenarios and shows that, after calibration, we are able to achieve a similar behavior in the simulation compared to the real-world. The UDP goodput revealed a deviation below 3 % and the corrected latency results deviate about 13 %, where, for the latter, we identified further potential for improvement. Furthermore, it would be interesting to validate our approach in more complex scenarios including more stations.

The memory and computation overhead of our framework is in the same complexity class as a corresponding simulation that is realized purely by ns-3. With our approach, however, we have the benefits of code-transparency, i.e., a common code-base for prototypes and simulation, and a strong correlation between both domains. In a small case study, we show the ability of our framework to find implementation defects in the wireless prototype code by

[2] http://warpproject.org/forums/viewtopic.php?id=3201

[3] http://warpproject.org/forums/viewtopic.php?id=3206

[4] http://warpproject.org/forums/viewtopic.php?id=3217

[5] http://warpproject.org/trac/wiki/802.11/Changelog

detecting bugs in the 802.11 Reference Design. We expect that with more systematic approaches, e.g., with symbolic execution, we will find further defects, which will strongly increase the reliability and correctness of prototypical deployments.

To summarize, our framework will not replace the development and testing of wireless communication protocols on SDR platforms, as the quality of its results depends on a strong correlation to the real world. However, it is a powerful tool to seamlessly switch from the real-world deployment to the simulation domain and vice versa. It thus complements results from a real-world deployment by enabling scenarios that were previously not possible due to time and cost constraints and, finally, it eases the verification process for wireless communication protocols.

## Acknowledgments

The research leading to these results has received funding from the German Research Foundation (DFG) under Agreement n. 625799 (MemoSim) and from the European Research Council under the EU's Horizon2020 Framework Programme / ERC Grant Agreement n. 647295 (SYMBIOSYS).

## 9. REFERENCES

[1] A. Varga. OMNeT++ Discrete Event Simulator. [Online] https://www.omnetpp.org.

[2] N. Anand, J. Lee, S. J. Lee, and E. W. Knightly. Mode and User Selection for Multi-user MIMO WLANs Without CSI. In *IEEE Conf. on Computer Communications (INFOCOM)*, pages 451–459, Apr. 2015.

[3] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using Static Analysis to Find Bugs. *IEEE Software*, 25(5):22–29, Sept. 2008.

[4] G. Braun, A. Nohl, A. Hoffmann, O. Schliebusch, R. Leupers, and H. Meyr. A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 23(12):1625–1639, Dec. 2004.

[5] L. Breslau, D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu, and H. Yu. Advances in Network Simulation. *IEEE Computer*, 33(5):59–67, June 2000.

[6] W. Choi, H. Lim, and A. Sabharwal. Power-Controlled Medium Access Control Protocol for Full-Duplex WiFi Networks. *IEEE Trans. on Wireless Communications*, 14(7):3601–3613, July 2015.

[7] Direct Code Execution Project. ns-3 Direct Code Execution (DCE) Manual, Release 1.8. Technical report, Mar. 2016.

[8] Free Software Foundation. The GNU Project Debugger. https://www.gnu.org/software/gdb/.

[9] P. Gallo, D. Garlisi, F. Giuliano, F. Gringoli, I. Tinnirello, and G. Bianchi. Wireless MAC Processor Networking: A Control Architecture for Expressing and Implementing High-Level Adaptation Policies in WLANs. *IEEE Vehicular Technology Magazine*, 8(4):81–89, Dec. 2013.

[10] IEEE. Standard for Information Technology–Telecommunications and Information Exchange Between Systems Local and Metropolitan Area Networks–Specific Requirements Part 11. *IEEE Std 802.11-2012*, pages 1–2793, Mar. 2012.

[11] M. M. H. Khan, H. K. Le, H. Ahmadi, T. F. Abdelzaher, and J. Han. Dustminer: Troubleshooting Interactive Complexity Bugs in Sensor Networks. In *Proc. of the 6th ACM Conf. on Embedded Netw. Sensor Systems*, pages 99–112, Nov. 2008.

[12] J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[13] S. Kraemer, L. Gao, J. Weinstock, R. Leupers, G. Ascheid, and H. Meyr. HySim: A Fast Simulation Framework for Embedded Software Development. In *5th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 75–80, Sept. 2007.

[14] M. Lacage. *Experimentation Tools for Networking Research*. PhD thesis, Université de Nice-Sophia Antipolis, 2010.

[15] M. Lacage and T. R. Henderson. Yet Another Network Simulator. In *Proceeding from the Workshop on ns-2: The IP Network Simulator*. ACM, Oct. 2006.

[16] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In *Proc. of the 1st Int'l Conf. on Embedded Networked Sensor Systems*, pages 126–137. ACM, 2003.

[17] Mango Communications. 802.11 Reference Design. [Online] http://mangocomm.com/802.11.

[18] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Cross-Level Sensor Network Simulation with COOJA. In *Proceedings of the 31st IEEE Conference on Local Computer Networks*, pages 641–648, Nov. 2006.

[19] J. Pelkey and G. Riley. Distributed Simulation with MPI in Ns-3. In *Proc. of the 4th Int'l ICST Conf. on Sim. Tools and Techn.*, SIMUTools '11, pages 410–414. ICST, Mar. 2011.

[20] Rice University and Mango Communications. The WARP Project. [Online] https://www.warpproject.org/trac.

[21] A. Schumacher, M. Serror, C. Dombrowski, and J. Gross. WARPsim: A Code-Transparent Network Simulator for WARP Devices. In *IEEE 16th Int'l Symp. on a World of Wireless, Mobile and Multimedia Networks*, June 2015.

[22] V. Sklyarov. Reconfigurable Models of Finite State Machines and Their Implementation in FPGAs. *Journal of Systems Architecture*, 47(14-15):1043–1064, Aug. 2002.

[23] H. Tazaki, F. Urbani, and T. Turletti. DCE Cradle: Simulate Network Protocols with Real Stacks for Better Realism. In *Proc. of the 6th Int'l ICST Conf. on Simulation Tools and Techniques*, pages 153–158, Mar. 2013.

[24] The ns-3 Consortium. ns-3 Discrete Event Network Simulator. [Online] https://www.nsnam.org.

[25] T. Ulversoy. Software Defined Radio: Challenges and Opportunities. *IEEE Communications Surveys Tutorials*, 12(4):531–550, Apr. 2010.

[26] A. Varga. The OMNeT++ Discrete Event Simulation System. *Proc. of the Europ. Sim. Multiconference (ESM)*, 2001.

[27] J. Wang, J. Beu, R. Bheda, T. Conte, Z. Dong, C. Kersey, M. Rasquinha, G. Riley, W. Song, H. Xiao, P. Xu, and S. Yalamanchili. Manifold: A Parallel Simulation Framework for Multicore Systems. In *IEEE Int'l Symp. on Perf. Analysis of Systems and Softw. (ISPASS)*, pages 106–115, Mar. 2014.

[28] D. Weber, J. Glaser, and S. Mahlknecht. Discrete Event Simulation Framework for Power Aware Wireless Sensor Networks. In *5th IEEE International Conference on Industrial Informatics*, volume 1, pages 335–340, June 2007.

[29] K. Wehrle, M. Güneş, and J. Gross, editors. *Modeling and Tools for Netw. Simulation*. Springer Berlin Heidelberg, 2010.

[30] H. Yu, O. Bejarano, and L. Zhong. Combating Inter-cell Interference in 802.11ac-based Multi-user MIMO Networks. In *Proc. of the 20th Int'l Conf. on Mobile Computing and Networking (MobiCom)*, pages 141–152. ACM, Sept. 2014.