Optimization of a novel WLAN Simulation Framework for Prototyping Network Applications and Protocols

Benjamin Beichler, Michael Rethfeldt, Hannes Raddatz,
Björn Konieczek, Peter Danielis, Christian Haubelt, Dirk Timmermann
University of Rostock
Institute of Applied Microelectronics and Computer Engineering
18051 Rostock, Germany, Tel.: +49 381 498-7278
Email: benjamin.beichler@uni-rostock.de

Abstract. Over the last few years, various types of wireless local area network (WLAN) infrastructures have been developed and established, taking into account the requirements of different kinds of applications and features, like scalability, robustness, energy-efficiency, and flexibility. The development of software applications, which depend on collaborating services on nodes, becomes more challenging, especially, due to an increasing number of device classes leading to heterogeneous architectures of communication nodes in wireless networks. In this paper, we present several optimizations of *ViPMesh*, a novel virtual prototyping framework for applications and protocols in IEEE 802.11 networks. We identified limiting issues regarding the guest VM boot time and frame exchange mechanism between guest VM and host system. By implementing a dynamic approach to switch the alternative clock source, we reduce the boot time to a reasonable level. Our second improvement introduces a shared memory buffer as a replacement for virtual serial channels and thus, the precision and performance of the simulation is significantly enhanced. Moreover, we add the capability to manage the emulation of nodes with different instruction set architectures, creating a testing and development environment for applications in wireless networks with heterogeneous platforms.

1. Introduction

Over the last few years, various types of wireless local area network (WLAN) infrastructures (such as centralized, ad hoc, and meshed) have been developed and established, taking into account the requirements of different kinds of applications and features, like scalability, robustness, energy-efficiency, and flexibility. With this move into new fields of applications and the further development of electronics, the number of device classes (e.g., smartphones, tablet computers, sensor nodes) has increased as well. Today, a more heterogeneous set of communication nodes is present in wireless networks. Moreover, the development of software applications, which depend on collaborating services on nodes with different hardware architectures, becomes more challenging. Without an appropriate testing and development environment, it is hard to comply with the need for a decreasing time-to-market of consumer hardware and software. Thereby, we pursue the evaluation of network applications as well as the prototyping of own optimization algorithms for IEEE 802.11 WLAN communication protocols. However, the setup of real-world test beds

is costly, impracticable, or simply not possible for increasing network sizes and dynamics. Network emulation or simulation allow for reproducible measurements and for flexible setups with a large number of nodes [8]. Above all, emulation permits software design and analysis on top of an unmodified protocol stack to leverage the comparability with real-world test beds. However, it still needs to be combined with simulations that account for wireless channel effects, different propagation environments, or dynamic network topologies due to node mobility. Furthermore, the integration of comprehensive simulation models is desirable to investigate the influence of modern WLAN technology parameters, such as IEEE 802.11 MIMO techniques, or high throughput (HT) configurations. As a result, the considerable computational effort of complex simulation models leads to the requirement of synchronizing both simulation and system time of emulated network participants.

We denote the combination of node emulation and network simulation as a *virtual prototyping* approach for the evaluation of applications and algorithms for WLAN networks. In [5], we proposed a framework for virtual prototyping of homogeneous WLAN networks, which combines a full system virtualization provided by QEMU and Linux container virtualization for executing real application software with an extended simulation solution called *wmediumd* for modeling radio channel effects. The simulation of multiple nodes in [5] was achieved exclusively by adding more containers within one virtual machine, which naturally need to share the same software instruction set. Therefore, this paper presents necessary optimizations of different parts to enable the existing framework to perform a more precise and faster simulation. This is achieved by employing technologies like shared memory and high resolution timers. Furthermore, additional changes to enable support of multiple instruction set architectures are introduced.

The remainder of this paper is organized as follows: In Section 2, we provide an overview of previous work in the field of combined simulation/emulation. Section 3 introduces the WLAN stack infrastructure of the Linux kernel and is followed by an overview of the *ViPMesh* framework in Section 4. Afterwards, Section 5 describes the revealed issues in detail and presents our proposed optimizations. Finally, we conclude our work in Section 6.

2. Related Work

In [7], the authors propose an approach to include real implementations of TCP/IP stacks and applications in a wireless network simulation. By using virtual machines (VMs) that are controlled by the event calendar of the network simulation process, a continuous time wall clock is not required but event-driven simulations are run. Thereby, simulations can be run that are slower or faster than real-time. For the implementation of the simulation framework proposed in [5], we adopt this concept of time decoupling. In [8], the authors extend their approach and present VMSimInt, which is a framework that integrates VMs into a network simulation tool to provide realistic OS behavior. Thereby, the focus is on providing a realistic TCP implementation. In their approach, each node is placed in its own VM, so the number of nodes corresponds to the number of required virtual machines. As opposed to our approach of isolating several nodes sharing the same instruction set architecture (ISA) in a single VM, using lightweight nested container virtualization, the concept of [8] does not scale well in terms of memory requirements as well as communication overhead. In their design, the addition of several nodes requires the same number of additional VMs. A virtual time system for OpenVZ-based network emulations is presented in [9]. The authors modify OpenVZ and its schedulers to be able to provide VMs each with their own virtual time, running on a single OS. As opposed to heavy-weight systems like Xen whose VMs contain both OS and appli-

cation, this approach scales better with an increasing number of VMs. VMs and their virtual times are managed by a control application running on the host OS and simulating a network of choice. The approach currently solely features container-based virtualization for Linux but prospectively the authors aim at developing a virtual time system for QEMU to support a larger number of platforms. In contrast, we combine Linux containers with a time-controlled QEMU-based system virtualization. Moreover, we integrated comprehensive physical-layer simulation models for IEEE 802.11 WLAN networks.

The work [6] addresses the problem of time divergence in hybrid network emulation by introducing a system called TimeSync that uses discrete-event simulation time to control and synchronize time advance on VMs. The core idea of TimeSync is to create a simulator-driven virtual timeline in the VMs participating in emulation. Although the core idea is similar to our approach, the focus of TimeSync is different as it uses stationary nodes connected by a wired Ethernet network rather than supporting IEEE 802.11 WLAN networks consisting of both stationary and mobile nodes.

In summary, the WLAN simulation framework in [5] improves the state-of-the-art by a first approach that complements a real protocol stack, as applied in practical systems, with a set of comprehensive simulation models that allow for the early design evaluation of real applications and algorithms in WLAN networks with IEEE 802.11 MIMO techniques, multi-channel operation, and mobility. However, the proposed implementation is limited to a single instruction set architecture for all communication nodes and provides limited simulation performance.

3. WLAN Infrastructure of the Linux Kernel

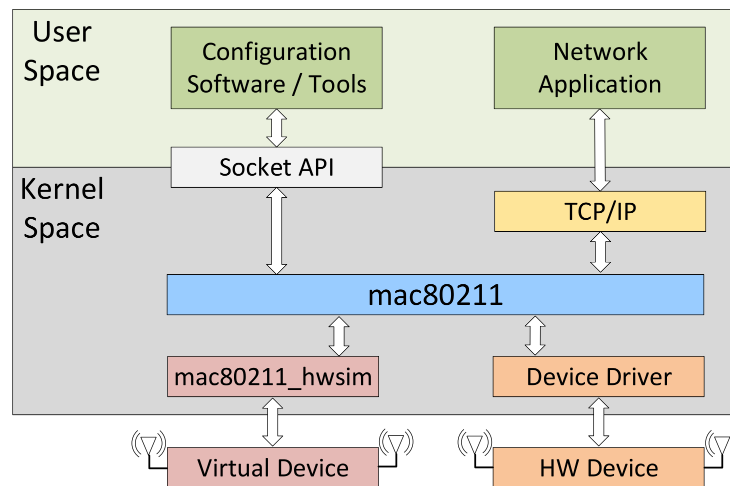

Figure 1: Linux kernel with mac80211 and mac80211_hwsim

The Linux kernel includes a sophisticated infrastructure mainly provided by the kernel module `mac80211` for the usage of IEEE 802.11 WLAN components, as depicted in Figure 1. Apart from the integration with the TCP/IP protocol stack, a socket API provides direct access for userspace software, e.g., to configure device or MAC layer parameters. From the view of a device driver, the module also provides a standardized interface to leverage code sharing for highly recurrent tasks in drivers for WLAN devices.

The module `mac80211` is used either by real WLAN hardware, or by emulated virtual WLAN devices, created with the help of kernel module `mac80211_hwsim`. In the basic mode, designed

for functional testing of `mac80211`, frames generated by `mac80211_hwsim` devices are assumed to be sent over a perfect channel, i.e., transmissions are always successful and issued immediately. Additionally, `mac80211_hwsim` implements an advanced mode, which provides an interface to forward encapsulated WLAN frames via a *netlink* socket. This socket interface mainly behaves like network sockets but is used for a request/response-based communication with the Linux kernel.

The solution presented in [5] uses this interface and therefore greatly extends the timing behavior simulation by applying a full medium access model and radio channel model to all transmitted frames while annotating resulting transmission latencies.

4. ViPMesh Framework Overview

The basic architecture of *ViPMesh* is shown in Figure 2. It is divided into an emulation part, based on a nested virtualization approach, and a simulation part modeling radio channel effects. Additionally, the time synchronization and data exchange between both parts are realized via virtual serial channels.

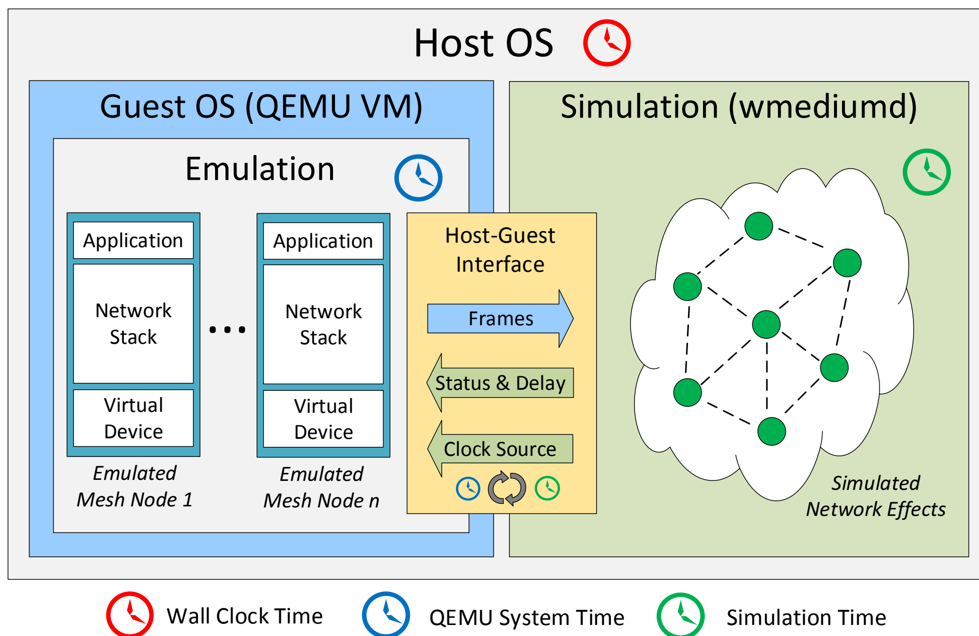

Figure 2: Architectural concept of ViPMesh

To virtualize the guest OS, building the emulation part of our framework, we rely on the open-source software QEMU [1]. This first virtualization step is needed to decouple wall clock and simulation time, as detailed in Section 5.1.

Inside the guest VM, we run an unmodified mainline version of the Linux kernel within a Debian distribution that emulates an arbitrary number of mesh nodes with real protocol stacks and applications. Using the same concept as of Linux containers (LXC), we assign each virtual device to a separate network namespace and thereby obtain isolated protocol stacks and applications while keeping nearly the same performance as without isolation, as shown in [3]. Consequently, we place every virtual WLAN device created by `mac80211_hwsim` into such an isolated environment.

On the one hand, this allows only the emulation of nodes with the same (or at least compatible) *Instruction Set Architecture* (ISA), but on the other hand the lightweight virtualization scales better

for large node counts. To explore setups with nodes of heterogeneous (or not compatible) ISAs, the framework architecture needs to be extended to create at least one QEMU VM per ISA.

On the guest VM, a program called `emuAdapter` is executed with real-time priority, which records timestamps, re-encapsulates all WLAN frames received from `mac80211_hwsim` and sends it via virtual serial connections provided by the QEMU framework to the host. Since `mac80211_hwsim` multiplexes all frames from all virtual interfaces, only one `emuAdapter` instance is needed per VM.

The framework's simulation part is realized in a daemon called `wmediumd` within the host OS, which is responsible for all radio channel effects, as shown in Figure 3. All models are applied in the shown order to each frame generated by the emulation, as described in [5]. Thereby, each model adds a virtual delay and/or attenuation of the signal-to-noise ratio (SNR) as well as bit errors to the simulated frame transmission.

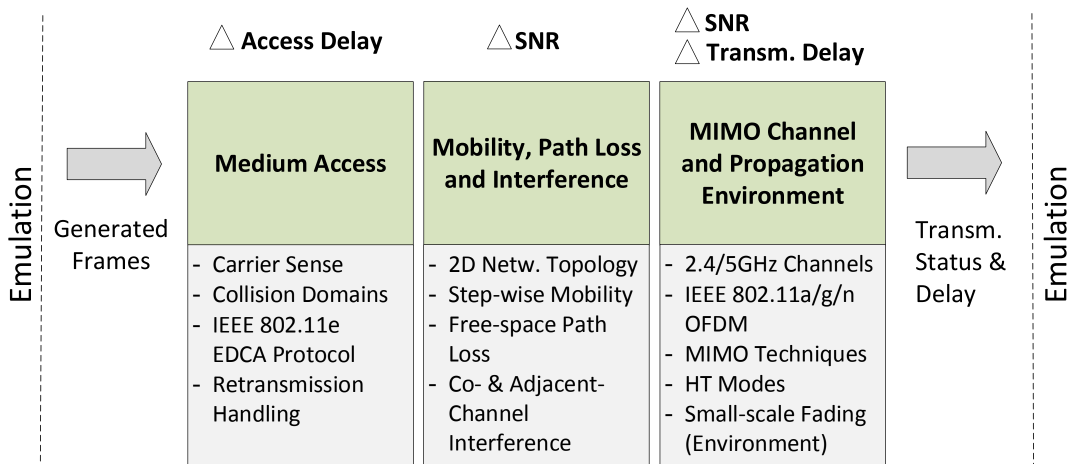

Figure 3: Simulation models of ViPMesh

Starting with a *Medium Access* model to derive the random back-off delay caused by contention-based channel access, SNR attenuation effects depending on spatial and spectral distance are determined by the *Mobility, Path Loss and Interference* model.

As a last step, technology-specific effects as well as scenario-specific multi-path fading characteristics are applied by the *MIMO Channel and Propagation Environment* model, and transmission status as well as overall delay are annotated in the frames and transmitted back to the emulation.

On this back path, the frames processed by the simulation are handled by the `emuAdapter` instance and forwarded to `mac80211_hwsim` after the annotated time in the QEMU VM has elapsed. Handling the communication with an userspace process has the advantage that the Linux kernel does not need to be modified for our simulation and could easily be exchanged or updated.

The simulation process (`wmediumd`) keeps track of all mesh nodes according to their position and other physical conditions and maps all incoming frames based on the MAC address of the transmitting mesh node. Beside the actual data exchange between `wmediumd` and `emuAdapter`, a separate communication channel for the time synchronization between simulation and QEMU instance is needed, which is described in section 5.1.

5. Detailed Problem Identification and Proposed Optimizations

The following sections focus on the description of identified issues within the *ViPMesh* framework and present our solutions. Our improvements focus on:

- the speedup of the guest VM boot time
- a more reliable and efficient data transfer between guest OS and simulation daemon
- extension of the simulation interfaces to multiple QEMU VMs with heterogeneous ISAs

5.1. Decoupling of Wall Clock and Simulation Time

To prevent the computing performance of the host system from affecting the combined emulation/simulation process, a decoupling of wall clock and simulation time is necessary. We adopt an approach originating from Werthmann et al. [8], comprising the control protocol *IKR SimLib* along with a corresponding patch for the QEMU emulator.

The original idea of the QEMU patch of *IKR SimLib* bases on an alternative clock source for the QEMU emulation. This clock source is used to emulate the precise time measurement devices of an emulated system. In the case of an x86-PC, this is the programmable interrupt controller (PIC) or, more recent, the advanced programmable interrupt controller (APIC). The emulated interrupt controllers within QEMU serve a standardized interface for target code and post the corresponding timed events to the inherent platform-independent timing infrastructure of QEMU. The standard QEMU clock source looks up the most recent event and uses the POSIX alarm signaling system of the host operating system to measure time. If an alarm signal is raised, the corresponding interrupt will be triggered and the execution of the simulated CPU is interrupted. The alternative clock source of *IKR SimLib* instead is keeping a list of all pending events and waits for a state of the CPU, in which all processes are blocked or paused by waiting on I/O requests or timed events.

Figure 4 shows the process flow of the guest and host interaction while using the alternate clock source. We illustrate the discrete advance of simulation time (green) and guest system time (blue) as arrows whereas the host OS wall clock time (red) is displayed as continuous timeline. Additionally, the processing times needed by emulation and simulation, as related to the wall clock time without influence on simulation and guest time, are expressed as bars.

When an idle state is reached, a message containing the current simulation time and the absolute timestamp of the next event of the clock source is transmitted over a unix pipe to the simulation process on the host system (solid arrow named "Waiting" in Figure 4). The simulation process will then respond a message with the new simulation time, which is greater than the received simulation time but smaller than or equal to the transmitted next time event (solid arrow named "Advance Time" in Figure 4). Afterwards, QEMU sets this new simulation time to its alternate clock source, resumes CPU execution and raises the corresponding interrupts. In this execution model, the QEMU VM in cooperation with the simulation daemon will act as discrete-event simulator.

A side effect of this behavior is the removal of the influence of computation time within the guest OS. In theory, every computational task could be processed between two timed events of the QEMU system. This could lead to problems with code, which is executed in an indefinite loop without a proper blocking condition. Therefore, the *IKR SimLib* defines an experimentally determined upper bound of executed blocks of translated target code, which also triggers a pause of the CPU emulation.

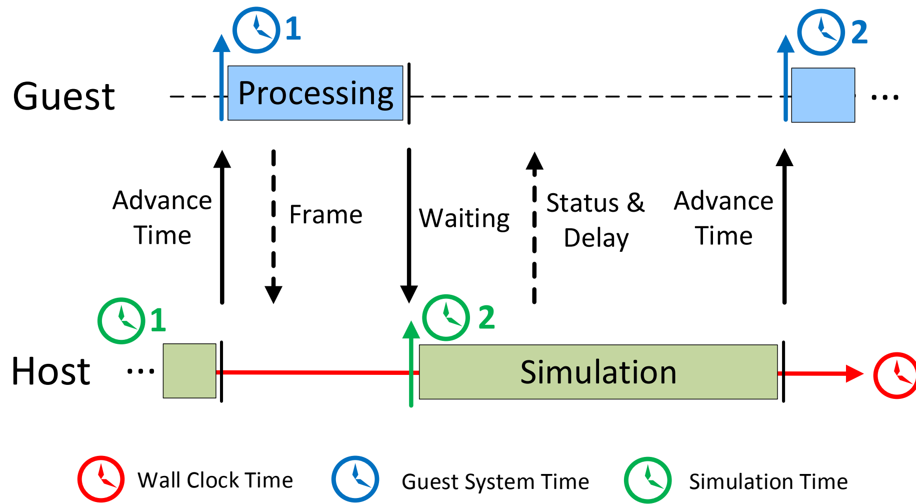

Figure 4: Emulation/Simulation Procedure of ViPMesh

The original QEMU is emulating a target architecture on a best-effort basis, i.e., after translation from target code to host code, no time annotation is done. Therefore, depending on the host performance, a simulated target could be faster or slower than a real target platform and no realistic performance estimation is possible. Removing any influence of the host performance on the target execution time emphasizes the impact of timings of the communication technology, channel effects, and network topology and suppresses influences of processing times. Moreover, we assume these network effects to dominate the overall communication latency.

Boosting VM Boot Time

The proposed modification by *IKR SimLib* has the downside, that in their work specialized kernels were used. Especially, the Advanced Configuration and Power Interface (ACPI) of the Linux kernel was deactivated in former works. Unfortunately, this prevents the Linux kernel from using high precision time sources, which need to be discovered via ACPI. On the other hand, the ACPI and connected Peripheral Component Interconnect (PCI) discovery function, extensively use the interrupt controller. This manifests in a small step size of, e.g., 10 ns in our emulation environment, for several seconds of simulation time, which results in a huge amount of wall clock time needed by the simulation.

On the one hand, the overhead introduced by communicating each time synchronization step through an operating system pipe as well as the alternating activation of simulation thread and QEMU VM thread, heavily increases the boot time of the QEMU VM. This results in a large delay on every start of the *ViPMesh* framework and strongly degrades its practical usage. On the other hand, without the high precision event timer (HPET) of the Linux kernel, it is impossible to establish a simulation with nearly real network latencies. A decreased precision of waiting functions in the C-API is caused by the absence of the HPET. This leads to inaccuracies of several 100 μs, which is the same order of magnitude as a complete transmission of a WLAN frame. Thus, the version of *ViPMesh* in [5] is negatively affected by this precision loss as the frame delivery often happens later as given in the time schedule determined by the simulation.

To circumvent this problem, we extended the alternative clock source from the *IKR SimLib* with the possibility to change the behavior of the alternative clock source at run-time. During boot up of the QEMU VM, the alternative clock source uses the same infrastructure as the traditional QEMU

event system and posts alarm signals to the operating system. When the QEMU VM is booted and an userspace shell is ready, the simulation daemon sends a newly defined message via the control pipe and the callback queues the request for activating the alternative clock. If the system gets into a suspended state, QEMU will consistently change the clock source.

This allows us to boot and setup the guest VM with the normal performance of a stock QEMU in several minutes and then switch to our intended behavior, which slows down the execution but greatly increases precision.

5.2. Frame Exchange Mechanism between Guest VM and Simulation

In Figure 4 (see Section 5.1), the basic emulation/simulation procedure for a discrete advance of simulation time including a frame transmission is depicted, showing the relation between the simulation time of the host OS and the system time of the guest VM.

A frame generated by an emulated node of the guest VM is passed to the host OS, along with further transmission information required, such as the current data rate and WLAN channel. On the host side, the annotated frame travels through several simulation steps. Results of the simulation are an information message, denoting transmission status (success or failure) to the frame originator, and a message containing the actual frame for the destination node. Moreover, overall frame delay for transmitter and receiver is included in the messages and reported back to the emulation. The guest VM then applies the actual delivery of all successful frames to the emulated nodes, as determined by the simulation.

The original approach in [5] used serial channels for all data exchange between guest and host. Especially, the communication of data frames (arrows with dashed lines in Figure 4) turned out to be a bottleneck for the simulation performance, caused by input/output latencies of the emulated serial connection in the host system. This is mainly caused by the high data amount (WLAN frames with message size of up to 1.5 KByte and a target bandwidth of several MBit/s), which needs to be transmitted, in comparison to the relative small message (8 byte timestamp plus 8 byte header) of the time synchronization protocol (solid black arrows).

Another problem we observed is a latent deadlock situation caused by the design of the patched QEMU VM. When the simulation triggers a new time advance, it needs to wait for new frames from the guest VM. Without an extra flow control mechanism, it is not easily determinable, whether the `emuAdapter` process has sent frames via its output connection. So, introducing a flow control (e.g., the `emuAdapter` sends 10 concurrent frames to the simulation) tends to create a deadlock as the `emuAdapter` may block while sending all frames (e.g., any buffer of the virtual serial connection fills up). Afterwards, the simulation also blocks after receiving the "waiting" message in Figure 4. This results from the fact that it tries to read all announced frames from the VM. Introducing a thread for continuously reading on the serial line may alleviate this problem but adds notable synchronization effort to the simulation. Also, increasing all buffers of the serial connection may help but since the data is passed through several layers (`emuAdapter` process → guest OS → serial connection device → host OS → `wmediumd` process), it is not easy to configure this consistently. Therefore, we use another solution for host/guest communication presented in the following.

Improvement of Frame Exchange Mechanism

To create a more reliable and faster communication mechanism between the `emuAdapter` process and the simulation process, we propose the usage of a shared memory segment between these two entities. The work of [4] proposed a *nahanni* device mechanism to share memory segments

between a QEMU VM and the host system. On the host, the POSIX shared memory API is used for memory accesses. On the guest OS, the shared memory segment is accessible via a generic PCI device, which exposes a device memory region to the corresponding shared memory. This would allow for the implementation of a PCI device driver in kernel space or an userspace based PCI UIO driver, as described in [2]. However, for the current implementation, we use only parts of the generic PCI driver, which provides a special file in the `/sys/` filesystem, representing the actual memory region. Such a memory region file can be simply mapped into the userspace virtual memory of the `emuAdapter` using the `mmap()` systemcall. This is possible because one of the main responsibilities of a PCI UIO driver is to serve as an easy interface for interrupt callbacks, but our shared memory implementation, described in the following paragraph, can act without such a signaling mechanism.

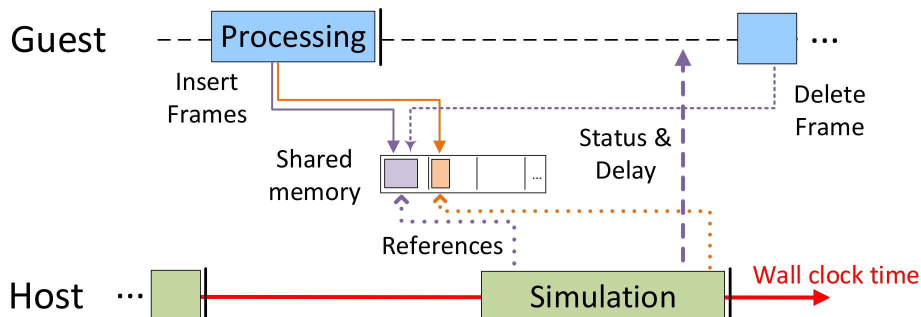

Figure 5: Usage of Shared Memory in ViPMesh

In Figure 5, a more detailed view on the frame exchange between guest VM and simulation process is given. The shared memory is split into a static array of WLAN frames with maximum size (including additional header and processing information), e.g., $1024 \cdot 1600\ byte$. If the `emuAdapter` process receives a new frame from the guest kernel, a free slot in the array is chosen and the corresponding frame is written to the shared memory. When the guest VM suspends its execution, the simulation iterates over all shared memory slots, searching for new frames and processes them. After that, all receivers are notified about a new frame by handing over a reference to the shared memory. Finally, the transmitter is informed about the transmission status and the frame data is deleted by the `emuAdapter` from the shared memory.

Due to the mutually exclusive execution of guest VM and simulation process, there is no write lock or semaphore needed for the shared memory. However, since the guest VM could be preempted while writing to the shared memory (caused by the defined upper bound of execution blocks, see Section 5.1), we need to ensure that the simulation only considers consistent frames, already completely written. This is done via an atomic write of a flag to the shared memory, which will be evaluated by the simulation.

Besides the saving of IO operations to deliver frames to the simulation, the shared memory enables a nearly zero-copy implementation of all frame analysis and forwarding functions and enhances simulation performance. Although a similar shared memory could be used also for the backward delivery of status information to the guest VM (dashed lines upwards), we did not identify this as a significant bottleneck and leave this as a subject for future optimization. Also, the search for unprocessed frames currently checks every entry for a valid frame and could be improved by smart reference structures to search only in important slices of the shared memory.

5.3. Extensions for Heterogeneous Simulation

The architecture of the *ViPMesh* framework in [5] shows the general possibility to extent the approach to manage multiple QEMU instances. However, as we have shown in this paper, the original architecture performs mediocre with the set of virtual serial channels of one QEMU VM. An upscaled approach with several new virtual serial channels is expected to perform proportionally worse. Therefore, the already proposed optimizations are necessary to allow a practical use and extension of our prototyping framework.

Obviously, the simulation process needs additional extensions to the management of time events, since not only one QEMU VM sends its next scheduled activation point but rather multiple VMs. Due to the architecture of the *IKR SimLib* patch, we are allowed to always set the simulation time between the current time and the next activation point. Consequently, we only need to take the minimum value through all activation points of all QEMU machines. The problem of keeping the simulation time of all QEMU machines synchronous has already been solved as well, since the simulation process always sends the same simulation time to all VMs concurrently.

Upscaling the shared memory for multiple VMs seems to be the easiest way to create multiple shared memories and keep the current implementation on the guest side. However, on the one hand, this will break the zero-copy approach because if a frame is sent from node `A` on VM `1` to node `B` on VM `2`, it will be saved in the shared memory of VM `1` and a reference to the frame is not sufficient to deliver the frame on VM `2`. On the other hand, one shared memory between all VMs breaks the assumption of only one writing process existing per time, when all VMs actually execute in parallel.

Therefore, we propose an additional modification of the management of the shared memory. After a proper configuration of all nodes, we presume that every MAC address simulated by a `mac80211_hwsim` device will only exist once over all VMs. This assumption inhibits the occurrence of MAC address collisions. Despite the fact that MAC addresses should be collision-free by definition, it may be interesting to investigate the robustness of an implementation according to MAC address collisions. This is usually not the focus during the development of network applications, and thus allows us to subdivide the shared memory into exactly one region for every emulated node. Other mechanisms, which hide the permanent MAC address of a transmitting node, e.g., anonymous WLAN probing, are not affected, since `mac80211_hwsim` always adds the unique ID of a transmitting node to a corresponding frame.

6. Conclusion

In this paper, we presented several optimizations of our virtual prototyping framework *ViPMesh* for applications and protocols in IEEE 802.11 networks. *ViPMesh* relies on WLAN interface emulation and QEMU-based system virtualization with nested container isolation to support the early design analysis of real software implementations on top of an unmodified network stack and OS. Adopting an alternative time source approach for QEMU, *ViPMesh* acts as discrete-event simulator. Furthermore, it integrates comprehensive medium access, channel, and environment models with support for interference effects, IEEE 802.11 MIMO, multi-channel operation, and device mobility. We identified issues regarding guest VM boot time and the frame exchange mechanism between guest VM and host system, which have limited the practical operation of our proposed framework. By implementing a dynamic approach to switch the alternative clock source, we reduced the boot time to a reasonable level. Our second improvement introduced a shared memory buffer as a replacement for virtual serial channels, and thus the precision and performance of the

simulation was significantly enhanced.

Moreover, we added the capability to manage the emulation of nodes with different instruction set architectures, creating a testing and development environment for application development in wireless networks with heterogeneous platforms.

Acknowledgment

The authors would like to thank the German Research Foundation (DFG), RTG 1424 (MuSAMA) for their financial support.

References

[1] F. Bellard: *QEMU, a Fast and Portable Dynamic Translator*. In *Proceedings of the USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, USA, 2005. USENIX Assn.

[2] J. Corbet: *Uio: user-space drivers*, 2007. `https://lwn.net/Articles/232575/`.

[3] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio: *An updated performance comparison of virtual machines and Linux containers*. In *IEEE ISPASS '15*, pages 171–172, March 2015.

[4] A. C. Macdonell: *Shared-memory Optimizations for Virtual Machines*. PhD thesis, Edmonton, Alta., Canada, 2011, ISBN 978-0-494-89468-2. AAINR89468.

[5] M. Rethfeldt, H. Raddatz, B. Beichler, B. Konieczek, D. Timmermann, Ch. Haubelt, and P. Danielis: *ViPMesh: A Virtual Prototyping Framework for IEEE 802.11s Wireless Mesh Networks*. In *12th IEEE International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 688–694, NY, USA, 2016. ISBN 978-1-5090-0724-0.

[6] F. Sultan, A. Poylisher, C. Serban, J. Lee, R. Chadha, C. J. Chiang, K. Whittaker, Ch. Scilla, and S. Ali: *Timesync: Virtual time for scalable, high-fidelity hybrid network emulation*. In *IEEE MILCOM*, 2012.

[7] Th. Werthmann, M. Kaschub, Ch. Blankenhorn, and Ch. M. Mueller: *Approaches for evaluating the application performance of future mobile networks*. European Cooperation in the Field of Scientific and Technical Research, COST IC1004 TD (11), 1038, 2011.

[8] Th. Werthmann, M. Kaschub, M. Kühlewind, S. Scholz, and D. Wagner: *VMSimInt: A Network Simulation Tool Supporting Integration of Arbitrary Kernels and Applications*. In *Proceedings of the 7th International ICST Conference on Simulation Tools and Techniques*, pages 56–65. ICST, 2014.

[9] Y. Zheng and D. M. Nicol: *A Virtual Time System for OpenVZ-Based Network Emulations*. In *Principles of Advanced and Distributed Simulation (PADS), 2011 IEEE Workshop on*, pages 1–10, June 2011.