Towards Network Centric Development of Embedded Systems

S. Schürmans*, E. Weingärtner†, T. Kempf*, G. Ascheid*, K. Wehrle†, R. Leupers*
*Institute for Integrated Signal Processing Systems, RWTH Aachen University
†Distributed Systems Group, RWTH Aachen University

***Abstract*— Nowadays, the development of embedded system hardware and related system software is mostly carried out using virtual platform environments. The high level of modeling detail (hardware elements are partially modeled in a cycle-accurate fashion) is required for many core design tasks. At the same time, the high computational complexity of virtual platforms caused by the detailed level of simulation hinders their application for modeling large networks of embedded systems. In this paper, we propose the integration of virtual platforms with network simulations, combining the accuracy of virtual platforms with the versatility and scalability of network simulation tools. Forming such a hybrid toolchain facilitates the detailed analysis of embedded network systems and related important design aspects, such as resource effectiveness, prior to their actual deployment.**

I. Introduction

Over the past decade, embedded systems with communication features have become a pervasive reality. For example, cellular phones and wi-fi home routers are embedded systems with the core task of providing network access to end users. Many multimedia devices such as set-top boxes or portable media players implement communication features for the purpose of retrieving additional content from the Internet or for sharing media data with other users.

The actual design of embedded systems, especially their core architecture, is often carried out using *Virtual Platforms* (VPs), which are basically detailed simulators of hardware platforms and the corresponding software. Well-established tools in the area of hardware-software co-design, for example Virtutech Simics [1] or CoWare Platform Architect [2], enable the flexible composition of new system designs. These consist of IP (Intellectual Property) blocks, for example processor cores or communication architectures, as well as custom hardware modules. Virtual platforms simulate the hardware of the entire embedded system at its present design state. This enables the execution of actual software within these environments. Therefore it becomes possible to develop system software like device drivers or operating systems concurrently to the hardware design phase. Moreover, the ability to observe and influence the behavior of the simulated system arbitrarily, e.g., by setting breakpoints and possibly changing the state of the system in a reproducible fashion, facilitates the robust implementation of both system software and hardware. However, VPs are difficult to employ for the investigation of large network scenarios that involve many communication peers, as simulating the entire system hardware for every host node is not feasible due to the high computational effort.

Network protocols are mostly evaluated using distinct discrete event-based network simulation tools that are not based on SystemC. *Network simulators*, for example ns-2 [3], ns-3 [4], OPNET Modeler [5] or OMNeT++ [6], allow to conveniently model arbitrary network topologies with any imaginable set of protocols. Besides the high degree of flexibility, scalability is another advantage of this methodology. Recent network simulators are able to simulate thousands of nodes on customary hardware, because the nodes are modeled on a significantly higher level than on a virtual platform. A key property of network simulations is their tendency to strongly abstract from all hardware details of the nodes. For example, most network simulations do not model the end host behavior at all. In many cases the model implements only a subset of the respective protocol to provide basic functionality to the simulation. This makes it difficult to draw conclusions about the performance of actual implementations regarding network protocols or network applications using solely network simulations.

Embedded devices with networking functionalities often are resource constrained, for example in terms of available energy, system memory and CPU performance. In order to validate the resource effectiveness of corresponding hardware and software, the early analysis of such in the design cycle is vital. For this purpose, we propose the integration of virtual platforms with network simulations in this paper, aiming at the network centric design of embedded system software and corresponding hardware. We utilize the network simulation for the context provisioning to the VP. This way, it becomes possible to analyze the system behavior given stimuli that closely resemble real-world network scenarios. Moreover, the detailed system model of today's VPs makes it possible to overcome the modeling limitations of current network simulations, especially according to the end host behavior and related performance metrics, such as CPU utilization, bus load, energy consumption and memory usage. We discuss our integration concept in further detail in Section II, before we introduce our corresponding implementation based on the CoWare Platform Architect and ns-3 in Section III. We evaluate the feasibility and the applicability of this approach in Section IV, compare our work with related approaches in Section V and conclude this paper with our essential findings in Section VI.

II. Conceptual Design

We now describe the conceptual design of our approach (see Figure 1). It facilitates the integration of a discrete event-based network simulator with a VP that models any embedded system hardware. The actual integration of the virtual platform and the network simulation is carried out using two communication flows. First, we need to enable the data exchange between simulated hosts and the system modeled by the VP. Second, the run-time execution of both must be synchronized, as the VP and the simulation operate in two distinct timing domains. Otherwise a potential time drift may corrupt the obtained measurements, as neither the VP nor the network simulation always execute in a real-time

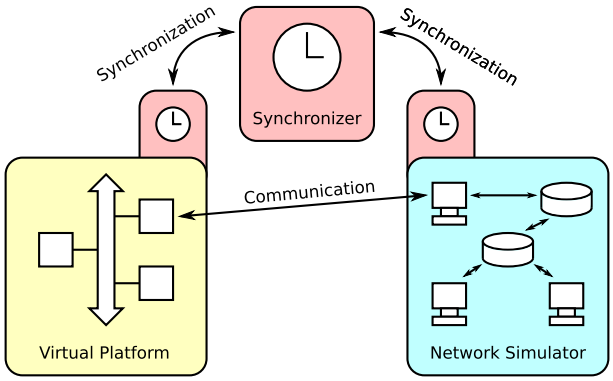

Fig. 1: Conceptual Design of Simulation Coupling

fashion. The synchronization of the embedded system modeled by the VP and the simulation is based on the provisioning of individual time quanta as earlier proposed in [7]: A central synchronization component, the so-called *Synchronizer*, issues discrete time quanta to the network simulator and the VP. Whenever the VP or the simulation completes its quantum, the execution is blocked until the synchronizer assigns the next quantum. In the following, we discuss important aspects regarding the VP and the simulation and how we integrate both to form a hybrid toolchain for both development and evaluation purposes.

A. Virtual Platform

A virtual platform [8] or virtual system prototype [9] defines a behavioral model of a system at various levels of abstraction. Typically, virtual platforms are utilized as an executable specification in order to support software and hardware development. The most important advantages are:

- Development and debugging of software before a hardware prototype is available. Debugging is non-intrusive and achieves typically better state visibility than debugging directly on the hardware.
- Analysis, optimization and verification of software and hardware.
- Easy modification and exploration capabilities reduce time and cost intensive hardware prototyping experiments.

Today the underlying technology of virtual platforms is based on SystemC [10] and the Transaction Level Modeling standard 2.0 (TLM-2) [11]. Especially the TLM-2 standard nicely reflects the common hardware design principle of component based design (CbD) [12]. Here a complete hardware platform is a collection of various IP components that are connected in order to execute a certain function, e.g. an entire wireless handset. Fundamental building blocks are processors (e.g. RISC, DSP), communication architectures (e.g. bus, crossbar) and components for data exchange like packet radios.

Since communication with the external environment can only occur over communication interfaces, whether wired or wireless, these are the natural choices to part the two simulation environments of virtual platforms and network simulators. In addition, the component based design paradigm allows to develop a virtual network adapter that looks to the virtual platform like a standard radio component, but on the other hand bridges the gap to the network simulation. Hence, in the course of this work the *NetChip* IP component based on SystemC has been developed, which is further discussed in Section III-A.

For synchronization of the virtual platform with the network simulator we developed the *SyncVP Component* (Section III-A). This standard SystemC component is a virtual device that can be easily integrated into every SystemC simulation and requires *no* connection or further hardware specific configuration. The component opens a channel to the synchronization server and blocks progression of the virtual platform simulation till a command to advance a time quantum occurs. This simple mechanism ensures synchronization with the network simulator introduced next.

B. Network Simulation

The task of the network simulator is to model the computer network to which the VP is connected. Respective simulation tools, for example ns-2 [3], ns-3 [4], OPNET [5] or OMNeT++ [13], model the communication of network hosts based on the paradigm of discrete event-based simulation. Essentially, basic communication primitives, such as sending network packets or the reception of data, are represented by events with an associated event execution time. All events are queued based on their time of execution and processed sequentially by the simulator. Hence, network simulators share many similarities with SystemC due to their internal event-based architecture.

For many of these simulators, large repositories of simulation models, such as channel models, protocol models or application models exist, and hence, one can often directly start setting up the simulation.

In order to facilitate the communication of the VP with the simulation, the network simulator needs to convert incoming packets to its own message representation. In a similar fashion, outgoing packets are serialized to the message format used by the VP. The synchronization of the VP with the network simulation is carried out using a custom event scheduler which blocks the simulation if the next event in the simulation queue is scheduled beyond the current time quantum.

III. Implementation

Following the discussion of our integration concept, we now delineate how we implemented the fusion of a VP and a network simulation into a modular workbench. As our VP integration is solely based on standard SystemC, hybrid VP network simulation setups can be executed using any SystemC simulation framework. Similarly, the network simulation is bridged with the VP using two generic and lightweight protocols based on UDP [14] for the exchange of communication data and synchronization messages. In the following, we lay out further details of our implementation and its core modules.

A. Virtual Platform Components

In order to integrate the VP into the joined setup, it needs to be interfaced with both the communication and the synchronization flows. We link the VP to these interfaces using two SystemC modules.

The first module, the so-called *SyncVP Component*, handles the synchronization of the VP with the network simulation. For this purpose, it simply needs to be added to a SystemC design of choice. As it does not provide any ports nor requires any connections to other SystemC modules, existing SystemC designs are not required to be changed in any other way. The SyncVP component carries out its task using only standard

```
void SyncVP_thread() {
  while (true) {
    do {
      msg = recv(SYNCHRONIZER);
    } while (msg.type != RUN_PERMISSION);
    wait(msg.quantum);
    send(SYNCHRONIZER, ACK);
  }
}
```

Lst. 1: Pseudocode of SyncVP component

SystemC features. As the pseudocode in Listing 1 illustrates, it executes the following four subtasks in an endless loop:

1) **Reception of a Run Permission Message**
   The synchronizer sends an UDP packet containing a run permission message whenever it assigns a new time quantum to the simulation. As SystemC is an extension of C, the SyncVP component can use a standard Berkeley Socket [15] to receive incoming messages.
2) **Suspending the Simulation**
   Before a time quantum has been assigned by the synchronizer, the entire simulation has to be halted. A common SystemC module can accomplish this by not returning control to the SystemC scheduler until the time quantum has been assigned. This way, the SystemC kernel does not know if any events will be scheduled before the code of the module returns control, so the kernel cannot execute any events after the current point in time. The SyncVP component executes blocking reads on the communication socket, which halts execution without busy waiting. Execution continues in the code of the SyncVP component when a run permission message is received.
3) **Running the Simulation**
   By putting itself to sleep for the duration of the assigned time quantum, the SyncVP component is able to advance the simulation. This is done by calling SystemC's `wait()` function with the duration of the quantum as parameter. All other parts of the simulation will continue to run for this time. Afterwards, the `wait()` function returns and control is passed back to SyncVP component.
4) **Acknowledgement of Time Slice Execution**
   After completing a time quantum, the SyncVP component has to inform the synchronizer. This is done by sending an UDP packet using the socket interface also used for reception of run permission messages.

The *Virtual Network Chip* implements the communication with the network simulator on the VP side. It has been modeled to work like a real network chip, except that the network link is modeled using an UDP based tunnel protocol for simulated Ethernet frames.

Figure 2 displays the internal structure of the network chip. In the VP environment, it provides a bus target interface and an interrupt initiator port. The bus interface enables the access to a memory buffer of 16kB and to four control registers: a *configuration register*, a *status register*, a *command register* and a *size register*. Incoming messages are not directly written to the message buffer in order to retain full control of the buffer via the bus interface. Instead, a reception is indicated in the status register. The command register can be used to make a received message available in the buffer and to send the buffer

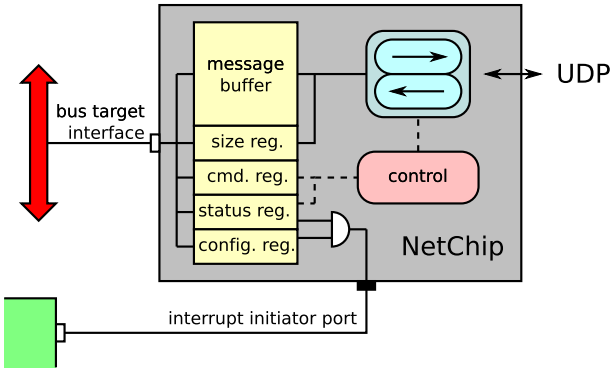

Fig. 2: Virtual Network Chip

contents as a new message. The configuration register allows configuring interrupts for the flags set in the status register.

B. Network Simulation

The integration of event-based network simulation tools with a VP essentially boils down to the implementation of a custom event scheduler (cf. [7]), as well as the realization of adequate methods for parsing VP network data and for the serialization of simulation messages.

At the present development stage, our modular workbench supports the integration of *OMNeT++* and *ns-3* based network simulations. While OMNeT++ provides the developer with a very convenient visualization of the network simulation, our current work focuses on ns-3 due to its increased flexibility, its off-the-shelf network emulation features and also due its growing importance in the network research community.

In order to integrate ns-3 with the VP, we introduce dedicated *ghost nodes* to the simulation. These ghost nodes are both connected to the simulation and to the VP. Essentially, the ghost nodes are placeholders for the VPs within the simulated network topology. If a simulated network packet arrives at the ghost node, it is wrapped into an UDP packet and tunneled to the corresponding VP. In a similar fashion, packets originating at the VP are converted to simulation packets upon their arrival at the network simulator. The traffic then is injected into the simulation at the respective ghost node.

IV. Evaluation

To show the capabilities of our proposed approach, we introduce a setup depicted in Figure 3. The simulated network consists out of two hosts connected via a single direct link. The first host is modeled within the network simulator, whereas the other one is a ghost node, whose functionality is simulated

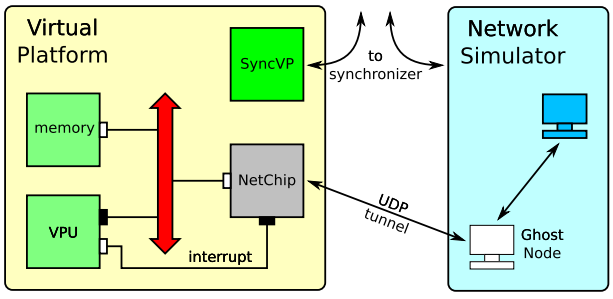

Fig. 3: Setup of VP and Network Simulator

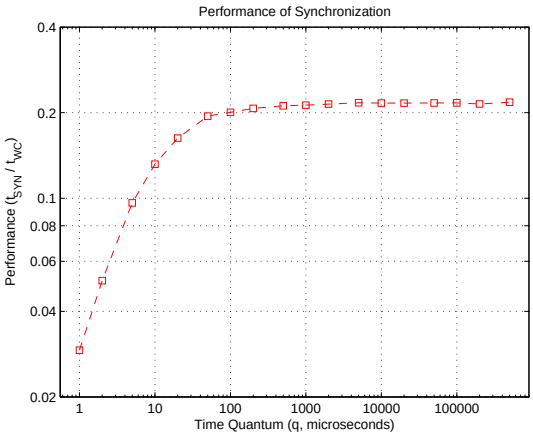

Fig. 4: Performance of Synchronization

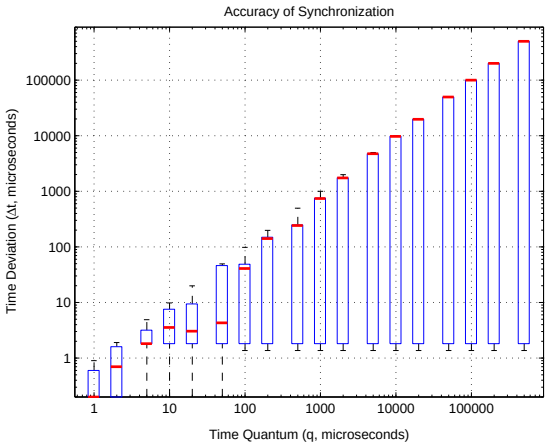

Fig. 5: Accuracy of Synchronization

on the VP. For synchronization, the VP utilizes a SyncVP Component and the rest of the system is build based on the Virtual Processing Unit (VPU) technology [16]. In this scenario, a single VPU instance represents an arbitrary RISC processor core, which is connected via an Advanced High-Performance Bus (AHB) clocked with 10 MHz to a memory subsystem and to the NetChip IP component, whose interrupt line is connected to the VPU.

The software executed on the VPU is a port of the micro IP (uIP) stack [17] extended with timing annotations to model realistic timing behavior of packet processing on an embedded processor core.

The host within the network simulator constantly sends ICMP echo request (*ping*) messages containing a payload of configurable length with a configurable interval between two consecutive messages. These messages are received by the uIP stack running on the VP, which replies with the corresponding ICMP echo reply message.

The setup used for evaluation consists of a version of ns-3 [4] extended with synchronization capabilities to run the network and CoWare Platform Architect [2] version 2009.1.1 to run the SystemC simulation. Both simulations are synchronized by the the mechanism described in [7]. All measurements have been performed on an AMD Athlon 64 X2 with 3GHz clock frequency and 6GB main memory running Scientific Linux 5 as operating system.

A. Performance and Accuracy

The size of the time quantum used for synchronization controls both performance and accuracy of the hybrid simulation. To measure the effect of time quantum size q on performance and accuracy, the test scenario is simulated with a variable time quantum ranging from $1\mu s$ to $500ms$. In each test configuration the host in the network simulator sends 100 ping requests with 56 bytes payload with an interval of $1s$ to be answered by the VP.

For measuring the performance, we relate the simulation time elapsed at the synchronizer t_{SYN} to the wall clock time required for running the simulation t_{WC}. The quotient t_{SYN}/t_{WC} over the quantum size q is graphed in Figure 4.

As two synchronization messages have to be exchanged between the synchronizer and both simulations for every time slot of quantum size q, the number of messages needed for a joint simulation of duration t_{SYN} equals $4{\cdot}t_{SYN}/q$. This is the reason for simulation performance to be worst for small quantum sizes q and to increase steadily towards longer quanta. For quantum sizes of $q \geq 10\mu s$, the steepness of the increase in performance starts to drop because the time needed for synchronization is getting smaller compared to the time needed for execution of the simulation for one time quantum. The performance is no more increasing for $q \geq 1ms$, because the synchronization overhead is negligible in relation to the simulation itself.

The accuracy of simulation coupling manifests in the observed time deviations between network simulator and VP. Every simulated network packet exchanged between network simulator and VP is seen once in the network simulator and once in the VP. We denote the time of a network packet entering/leaving the network simulator by t_{NS} and the time of it entering/leaving the VP by t_{VP}. Thus, the time deviation is $\Delta t := |t_{NS} - t_{VP}|$. We graph the distribution of the deviations Δt observed for the 200 network packets exchanged in the test case described above.

Figure 5 shows the median, the 25% and 75% percentiles as well as the minimum and maximum of the deviations Δt over the quantum size q.

The boundedness of Δt is the most important fact observable in the graph: $\Delta t \leq q$ holds for all quantum sizes q. This shows that the synchronization of simulations is working. For small quantum sizes $q \leq 2\mu s$, a lot of messages are observed in the network simulator and in the VP at the same time, the 25% pecentile is at the bottom of the figure. This is caused by the observed deviation being almost zero for a large percentage of packets. Exact timing still occurs occasionally up to $q = 50\mu s$ as can be seen from the minimum touching the bottom. However, the 25% percentile being shifted towards the maximum shows that most messages are already experiencing a time deviation. Starting from $q = 100\mu s$, most messages experience almost the full time deviation possible. Additionally, no message is arriving at the exact point in time any more. Both effects can be explained by looking at the simulation speed of the network simulator and the VP. As the most dominant timings in the network simulator are in the range of milliseconds, whereas the VP deals mostly with timings below microseconds, the VP has to process considerably more events per simulation time than the network simulator. The network

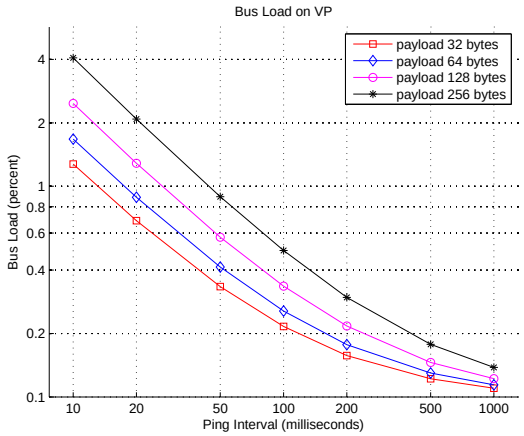

Fig. 6: Bus Load on Virtual Platform

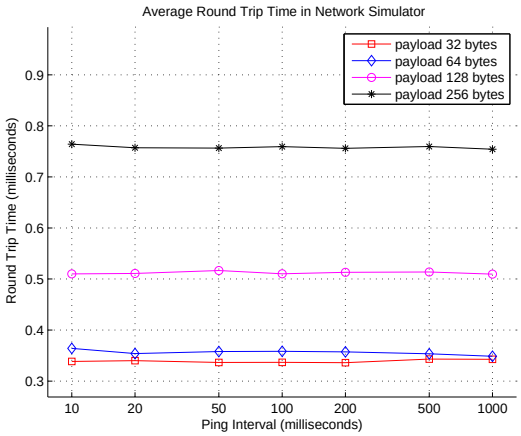

Fig. 7: RTTs observed by Node inside Network Simulation

simulator being significantly faster than the VP results in the VP having just started simulation of a time quantum when the network simulator has already completed it. Thus all packets sent from the network simulator to the VP will be seen on the VP at the beginning of the current time quantum and all packets sent from the VP to the network simulator will not be detected by the network simulator until begin of the next time quantum.

Performance (Figure 4) and accuracy (Figure 5) measurements show that a tradeoff exists between those parameters. With decreasing quantum size and thus increasing accuracy, the performance drops. While the increase in accuracy is uniform over the entire range of quantum sizes, the drop in performance is only marginal for large quantum sizes q and starts to become substantial for $q < 100\mu s$. Thus, a good tradeoff between performance and accuracy is achieved for a quantum size of $q = 100\mu s$.

B. Load on Virtual Platform

Our approach of simulation coupling is intended to help analyzing network induced effects on the VP. To illustrate this using a simple example, we measure the effects of incoming ping requests onto the load observed on the VP.

We first fix the time quantum used for synchronization to $q := 100\mu s$, which provides the best accuracy that can be obtained without significantly impairing performance. Instead of q, we now vary the ping payload length from 32 to 256 bytes and the ping interval from $10ms$ to $1s$. For every configuration, we measure the bus load of the VP, which is plotted over the length of the ping interval for the four different payload lengths in Figure 6.

With shorter intervals between the ping requests, the number of network packets to process increases, which leads to a higher bus load on the VP. For small ping intervals of up to $100ms$, the load is almost proportional to the number of packets to process per time unit and thus reciprocal to the length of the ping interval. For longer intervals, the load imposed by the ping is no more significantly larger than the background load on the VP unrelated to processing network packets. Thus, the decrease of the load is getting smaller for large intervals. The effect of the payload length is similar to the effect of the ping interval. With increasing payload size, the load rises. For small payload sizes of 32 and 64 bytes, doubling the size only leads to a growth in load of about one third. The reason is the constant load needed for reacting to a packet reception and for processing the header. In contrast, the bus loads for payload lengths 128 and 256 differ by almost a factor of two, because the header processing overhead is less predominant for those larger packet sizes.

As the VP was not modified or reconfigured during those measurements and all adaptions were only done to the network simulator, this example clearly demonstrates how network effects influencing the status of a platform can be analyzed using the proposed approach.

C. Round Trip Times

To show that the effects on the platform also impact the network, we measured the round trip times (RTTs) of the pings in the network simulator. These measurements were taken from the simulation runs also used for measuring the bus load on the VP. Instead of the load, Figure 7 shows the round trip time on the Y-axis of the diagram while keeping the interval on the X-axis and the different payloads.

The payload length influences the response time of the VP and thus the round trip time seen in the network simulator. Processing of longer packets takes longer as more data transfer has to happen between network chip and processing core and more computation has to be performed, for example for calculating the checksums of the reply. The round trip times show a dependency on payload length that is composed of a part proportional to payload length, which is dominant for the longer payloads of 128 and 256 bytes, and a constant part, which is dominant for the shorter payloads of 32 and 64 bytes. In contrast, the round trip time is almost independent of the ping interval. The reason is the round trip time being shorter than the interval. Thus, a ping request is fully processed and the VP is idle again before the next ping request arrives.

From these measurements we conclude that processing efforts on the VP in fact may influence sensitive network performance metrics, for example round trip times. Thus, the integration of the VP provides the network simulation with deeper insight into the timings to expect from the host. The inclusion of a host simulated in detail on a VP can hence help to improve the analysis of the network, as it allows to check if the timing behaviour of the hosts modeled inside the network simulator are realistic.

V. Related Work

Most Virtual Platforms are based on SystemC [10] and TLM-2 [11]. There are pure SystemC environments like [18] that can be used to run any SystemC simulation, but do not contain building blocks for a VP. CoWare Platform Architect [2] is a whole development environment for VPs. It provides a large library of SystemC/TLM-2 modules modeling processor cores, busses and memories. Next to a graphical editor used to assemble the VP from pre- and user-defined SystemC modules, a feature-rich SystemC debugging environment is available to trace and analyze the VP. There are also other VP products that are fully compliant to SystemC and TLM-2 and provide similar tools and module libraries, for example Synopsys Innovator [19]. In contrast, Virtutech Simics [20] and Carbon Design SoCDesigner [21] are two examples of Virtual Platform solutions not yet fully supporting SystemC, but including some interfaces to SystemC.

Virtutech Simics Network Simulation [22] is able to simulate networks of virtual nodes at different levels of abstraction while including network simulation as well as platform effects. However, the platform is not modeled in SystemC, which excludes the usage of components from most libraries and of many existing VPs.

A major influence for this work is the concept of *network emulation* as proposed by Fall [23], in which physical machines are connected to a event-driven network simulation. Nowadays, many simulators, most notably ns-3 [4], provide according features. All these approaches suffer from the problem of requiring the simulation to be real-time capable. In our previous work [7], we proposed a solution for this issue by introducing a synchronization scheme and by replacing the physical hosts with Xen based virtual machines. By contrast, the integration of VPs with network simulations allows the reproducible execution of code on a VP and provides the developer with a much higher possible degree of simulation accuracy.

VI. Conclusion

In this paper, we have proposed the integration of virtual platforms (VPs) for embedded systems development with network simulations. This novel hybrid approach facilitates the close analysis of embedded systems hardware and software during the design phase inside a large-scale simulated network context. The actual integration is based on two generic communication flows, one for data exchange and one for run-time synchronization purposes, which are realized using generic lightweight protocols. On the VP side, the plain SystemC implementation ensures a wide compatibility with existing SystemC environments. As indicated in our evaluation, integrating a network simulator with a VP is feasible with a reasonable accuracy while introducing only a slight overhead. Using this approach, we showed that the networking context of an embedded system directly impacts the load on the VP, for example due to a varying network load. In addition, we were able to demonstrate that the accurate modeling of VP timings also influences network performance metrics, for instance round trip times.

All in all, we conclude that integrating virtual platforms with network simulations extends the applicability of both, as they mutually benefit from the advantages of each other. We therefore regard this hybrid approach to be of major interest for researchers and developers in the domains of system design and computer networks.

VII. Acknowledgments

This work has been partially funded by the UMIC Research Centre and by the DFG Graduate School 643 at RWTH Aachen University.

References

[1] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002.

[2] "CoWare Platform Architect," [Online] Available http://www.coware.com/products/platformarchitect.php (accessed 08/2009).

[3] "ns-2 Website," [Online] Available http://www.isi.edu/nsnam/ns/ (accessed 07/2009).

[4] "ns-3 Website," [Online] Available http://www.nsnam.org/ (accessed 07/2009).

[5] O. Inc, "OPNET modeler," [Online] Available http://www.opnet.com (accessed 07/2009).

[6] A. Varga and R. Hornig, "An overview of the OMNeT++ simulation environment," in *SIMUTools 2008*, Marseille, France, March 2008.

[7] E. Weingärtner, F. Schmidt, T. Heer, and K. Wehrle, "Synchronized network emulation: matching prototypes with complex simulations," *SIGMETRICS Perform. Eval. Rev.*, vol. 36, no. 2, pp. 58–63, 2008.

[8] "CoWare Virtual Platform," [Online] Available http://www.coware.com/products/virtualplatform.php (accessed 08/2009).

[9] G. Hellestrand, "The revolution in systems engineering," *Spectrum, IEEE*, vol. 36, no. 9, pp. 43–51, Sep 1999.

[10] D. L. John Aynsley, *IEEE Standard SystemC Language Reference Manual*, IEEE Computer Society Std. 1666, Rev. 2005, March 2006.

[11] J. Aynsley, *OSCI TLM-2.0 Language Reference Manual*, Open SystemC Initiative (OSCI) Std. 2.0.1, Rev. JA32, July 2009.

[12] W. Cesario, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. Jerraya, L. Gauthier, and M. Diaz-Nava, "Multiprocessor SoC platforms: a component-based design approach," *Design & Test of Computers, IEEE*, vol. 19, no. 6, pp. 52–63, Nov/Dec 2002.

[13] A. Varga and R. Hornig, "An overview of the OMNeT++ simulation environment," in *SimuTools*, S. Molnár, J. Heath, O. Dalle, and G. A. Wainer, Eds. ICST, 2008, p. 60.

[14] J. B. Postel, "User datagram protocol," Request for Comments (RFC) 768, Aug. 1980.

[15] W. R. Stevens, B. Fenner, and A. M. Rudoff, *UNIX Network Programming, Vol. 1*. Pearson Education, 2003.

[16] Kempf, T., Dörper, M., Leupers, R., Ascheid, G. and H. Meyr (ISS Aachen, DE); Kogel, T. and B. Vanthournout (CoWare Inc., BE), "A Modular Simulation Framework for Spatial and Temporal Task Mapping onto Multi-Processor SoC Platforms," in *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, Munich, Germany, March 2005.

[17] A. Dunkels, "Full TCP/IP for 8-bit architectures," in *MobiSys '03: Proceedings of the 1st international conference on Mobile systems, applications and services*. New York, NY, USA: ACM, 2003, pp. 85–98.

[18] *OSCI SystemC Language and Examples*, Open SystemC Initiative (OSCI), March 2007.

[19] "Synopsys Innovator," [Online] Available http://www.synopsys.com/ (accessed 08/2009).

[20] *Virtutech Simics*, [Online] Available http://www.virtutech.com/ (accessed 08/2009).

[21] C. D. S. Inc., *http://carbondesignsystems.com/*.

[22] *Virtutech Simics Network Simulation*, [Online] Available http://www.virtutech.com/whitepapers/networking.html (accessed 08/2009).

[23] K. Fall, "Network emulation in the VINT/NS simulator," *Proceedings of the fourth IEEE Symposium on Computers and Communications*, Jul. 1999.