A Generic Functional Simulation of Heterogeneous Systems

Sebastian Rachuj(✉), Marc Reichenbach, and Dietmar Fey

Friedrich-Alexander University Erlangen-Nürnberg (FAU), Erlangen, Germany
{sebastian.rachuj,marc.reichenbach,dietmar.fey}@fau.de

Abstract. Virtual Prototypes are often used for software development before the actual hardware configuration of the finished product is available. Today's platforms often provide different kinds of processors forming a heterogeneous system. For example, ADAS applications require dedicated realtime processors, parallel accelerators like graphics cards and general purpose CPUs. This paper presents an approach for creating a simulation system for a heterogeneous system by using already available processor models. The approach is intended to be flexible and to support different kinds of models to fulfill the requirements of a heterogeneous system. Simulators should easily be exchangeable by simulators with the same architecture support. It was possible to identify the SystemC connection of the considered general purpose CPU models as a bottleneck for the simulation speed. The connection to the realtime core suffers from a necessary connection via the network which is evaluated in more detail. Combining the GPU emulator with the rest of the system reduces the simulation speed of the CUDA kernels in a negligible manner.

1 Introduction

The degree of automation in vehicles rises every year. There are already many different Advanced Driver Assistance Systems (ADAS) that help the driver and are even capable of taking full control of the car [7,8]. Providing the necessary performance and still allowing the safety-critical parts to get certified requires heterogeneous systems. These kinds of systems are already established in the realm of ADAS. They include general purpose processors, many core accelerators, and real-time processors. The Nvidia *Drive PX 2* is an example for a development board of a system that contains AArch64 compatible ARM cores, Nvidia Pascal GPUs, and an Infineon *AURIX* [16]. Audi proposes another platform called *zFAS* containing multi-core processors, reconfigurable hardware and specialized DSPs [2].

During the software development of new ADAS systems, the final hardware setup is usually not yet determined. Depending on the real-time requirements, kind of algorithm and necessary computing power, different characteristics of the final processing system have to be satisfied. For that reason, choosing the correct components and writing the software should be done cooperatively. This can be

M. Schoeberl et al. (Eds.): ARCS 2019, LNCS 11479, pp. 128–141, 2019.
https://doi.org/10.1007/978-3-030-18656-2_10

achieved by using virtual prototypes of the hardware that offer different levels of abstraction [11]. While very abstract emulation reaches high execution speeds, it suffers in accuracy of predicting the nonfunctional properties like required energy and runtime behavior. On the other hand, a very detailed simulation of the heterogeneous system might reach nearly real world values for the predicted values but is too slow for greater workloads as they might occur in ADAS algorithms.

Since a heterogeneous system contains multiple different kinds of processors (e.g. CPUs, GPUs, specialized ASICs, etc.), a virtual platform is required that also provides models for all of these components including their interconnections. A lot of processor simulators are available separately but can be connected to a SystemC runtime, a framework for implementing discrete simulations [10]. This allows the usage of already available models within virtual prototypes of heterogeneous systems.

The goal of this paper is to show how to combine multiple unrelated simulation models with the help of their SystemC bindings to create a mere functional simulation of a heterogeneous system as it might be used in current or future vehicles. Especially models that can be extended by means of determining nonfunctional properties are taken into account. However, their ability is not used yet. Another aim is to stay generic in a way that allows the inclusion and interchangeability of arbitrary SystemC compatible simulation models into the heterogeneous virtual platform and to avoid changes within the taken models. Hence, the approach was implemented with gem5 [3], OVP from Imperas, and the ARM Fast Models to simulate general purpose CPUs, GPU Ocelot [6] and GPGPU-Sim [1] to provide support for a CUDA-compatible accelerator core, and the AURIX model from Infineon to offer a realtime processor. After presenting the connection approaches, the simulation runtime performance impacts are identified and evaluated.

2 Related Work

Heterogeneous simulators are no new invention. Coupling virtual prototypes of general purpose processors with GPU emulation tools has been done before. A prominent example is *gem5-gpu* which also uses GPGPU-Sim and connects it to gem5 [13]. Power et al. created patches that modify the source code of the two simulators to allow the integration. They also took care about modeling the memory system including cache coherency protocols. Software can be run on the simulated GPU by using a wrapper for the CUDA runtime library. This enables the usage of available CUDA code but requires the binary to be linked to the wrapper before it can be deployed on gem5-gpu. In this paper a similar approach is presented that implements the coupling in a more generic way by offering memory mapped input and output registers. This allows not only gem5 to be used as a simulator for the general purpose CPU but also OVP and the ARM Fast Models. Still, a small software wrapper for the applications is required to exploit the simulated GPU.

A direct integration of a GPU simulation into gem5 was done by AMD. They added an accelerator model that is compatible to the GCN version 3 instruction set architecture and achieved an average absolute error of 42% [9]. Major difference to *gem5-gpu* and this paper is the supported runtime environment. AMD's approach is available to all languages supported by their HCC compiler including OpenCL, C++AMP, etc. while only CUDA and OpenCL is supported by GPGPU-Sim.

Further works include FusionSim (formerly on www.fusionsim.ca but not available anymore) and Multi2Sim [17] which both don't support SystemC coupling out of the box. Thus, they were not in line for connecting the real-time processor since that would have meant changes within the provided models. To the authors' knowledge, there is no generic coupling of processor models and many core accelerator models to realtime processor simulators available yet.

3 Virtual Prototype for Heterogeneous Systems

The proposed virtual prototype for heterogeneous systems uses the TLM library of SystemC for loosely timed simulations as its core because most available simulation models allow coupling with it. Figure 1 shows an overview of the system that was created for this paper. An arrow denotes a possible connection between an initiator socket (beginning of the arrow) and a target socket (end of the arrow). The prototype includes components often found on today's embedded ADAS platforms like general purpose CPUs, parallel accelerators and a dependable realtime CPU which is certified according to ISO 26262. There are already simulator models for these processors available. However, the connection to the realtime CPU and the linkage to the CUDA accelerators was newly created for this paper. Additionally, the central router which is the only strictly required part of the prototype and the peripherals had to be supplied. Excluding the bus, all models can freely be replaced or omitted allowing a generic adaption to the needs of the developed application. However, connecting the native simulators comes with an impact that is analyzed in Sect. 4.

3.1 General Purpose CPU

Most heterogeneous systems still contain a powerful general purpose CPU. Since the target application is an embedded system as it might be deployed in an ADAS application, the choice was to use the AArch64 instruction set architecture as a reference. Hence, it is sensible to consider the ARM Fast Models as an ARM instruction set simulator which allow a high simulation speed. Like the other processor models, it offers SystemC bindings and can easily be connected to the bus system. For this paper it was used in combination with the support libraries provided by Synopsys. Open Virtual Platforms (OVP) which is offered by Imperas is similar to the ARM Fast Models but also support many different instruction set architectures. It has already been used in research and was extended by runtime and power estimation functionality [5,15].

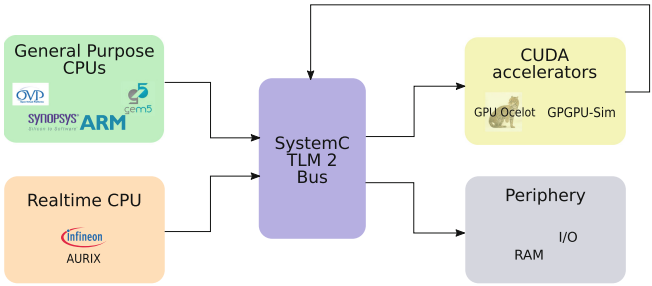

Fig. 1. The heterogeneous virtual prototype. General purpose CPUs can be provided by OVP, gem5 and the ARM Fast Models, the CUDA accelerator is implemented by GPU Ocelot or GPGPU-Sim. The real-time CPU is supplied by the Infineon AURIX model. Only the central bus is required.

Another simulation framework that was investigated for this paper is gem5 which supports different instruction set architectures and also offers multiple architecture backends. Available backends are the TimingSimple model implementing a single cycle CPU with the possibility to add a fine grained memory hierarchy. Additionally, the O3 model offers an out-of-order processor pipeline simulation which requires simulated caches to work correctly. In comparison to the previous two simulators, gem5 is much slower since it does not provide just-in-time compilation of the guest code. However, due to the detailed architecture description, a better runtime prediction can be achieved when using the detailed backends. The SystemC connection was established by Menard et al. who added a new slave type to gem5 allowing to interface with custom TLM targets [12]. Since the goal of this paper is to provide a heterogeneous virtual platform for functional emulation, the TimingSimple backend of gem5 was used. It allows adding a custom memory hierarchy but avoids an in-depth simulation of the microarchitecture. The generic approach presented in this paper allows all of these three general purpose CPU simulators to be chosen and integrated into the prototype. They can act as initiators of a TLM connection which makes it possible to directly connect them to the central bus system without further modifications.

3.2 GPU

Alongside the general purpose CPUs, an approach for emulating CUDA compatible accelerator cores was also accomplished. Parallel processors of this kind are very important for supporting computer vision applications like required for pedestrian or traffic sign recognition. There are two GPU simulators available that provide a CUDA runtime library to intercept the API calls and forward it to the backend. One of them is GPU Ocelot which implements a dynamic translation framework for translating PTX code into native machine code of the host CPU using LLVM [6]. To the authors' knowledge, it is not developed any

more. GPGPU-Sim, on the other hand, is a simulator for CUDA or OpenCL compatible GPUs which is still actively extended[1] [1].

The connection of the GPU simulators to the virtual prototype that had to be implemented for this paper was done by providing memory mapped input and output registers. They can be used to set the parameters of CUDA runtime functions and eventually to also call the function itself. Internally, arguments representing virtual addresses of the main memory are translated into global pointers of the SystemC instance which enable direct access to the underlying memory buffers. This is accomplished with the help of TLM's direct memory interface (DMI) that is used to request pointers from the central bus (compare the arrow from the CUDA accelerator back into the bus in Fig. 1). Delivering a pointer also requires the RAM implementation to support the DMI. Finally, the processed parameters are forwarded to the global CUDA runtime function available in the simulator. Depending on the library, the simulation binary is linked to, the functions of GPU Ocelot or GPGPU-Sim are used. It is even possible to use the real graphics card of a system by taking the standard CUDA runtime library deployed by Nvidia. This allows a Hardware-In-The-Loop approach which might be helpful for evaluation tasks with a fixed GPU architecture.

Another approach to integrate a GPU simulator implementing a runtime API into processor simulators is realized by *gem5-gpu* and the GCN3 implementation of AMD which use the Syscall Emulation (SE) facilities of gem5 [9,13]. However, this requires strongly simulator dependent code which should be avoided for the generic virtual prototype. OVP also supports adding additional syscalls by using the intercept library that allows the definition of callbacks when the requested syscalls are executed. But this method is not portable between different simulators and contradicts to the stated aim of this paper to offer a generic virtual prototype with exchangeable processor cores. Hence, this mode was not considered for the proposed platform.

3.3 Realtime Processor

The automotive industry always had a requirement for reliable and deterministic processor cores. As a result, specialized CPUs were created that offer distinct features like lockstep execution and the possibility to get accurate runtime predictions. Examples include the ARM Cortex-R and the Infineon TriCore families offering ISO 26262 compliance. Latter can be simulated by a so-called c-model that offers simulation models of an AURIX System-On-Chip. It contains multiple TriCores, common accelerators and bus transceivers for protocols often found in vehicles like CAN and FlexRay.

Due to platform restrictions of the involved models and their supported operating systems, it was not possible to run the whole heterogeneous system on the same machine within the same SystemC runtime environment. For this reason, a method for distributed SystemC simulation had to be implemented for this paper to enable a combined simulation of the realtime processor with the rest of

[1] As of February 2019.

the proposed prototype. It is loosely based on SystemC-Link [18] in the way that it uses latencies within the modelled design to reduce the experienced latency of the host network. To realize this connection, two major challenges had to be managed. First, a synchronization mechanism of simulation time was required to avoid one simulation instance to run ahead of the other one. Second, a possibility for data exchange had to established.

Synchronization can be done by periodically sending messages containing the current simulation time stamp of one SystemC instance to the other one. At the beginning of the simulation or after the last received foreign time stamp message, a time equal to the predefined latency can be simulated. If during this time another time stamp message is received, the simulation will execute with its maximal speed and no waiting times have to be introduced. This corresponds to the best case part of Fig. 2 where both simulators run their full speed. However, if one SystemC instance is faster than the other one, it will find out that the received time stamps lack far behind. When the difference between the local and the remote time gets greater than a predetermined threshold, the faster simulation will be paused until the difference got smaller again. This allows the both parts to be run with a resulting simulation speed, in terms of simulated seconds, of the slower participating simulation. If no further foreign time stamp message was received during the latency time, the simulation also has to be paused until new information about the other part arrived. This can be seen as the worst case part of Fig. 2 where the execution of both SystemC instances cannot resume until the new message is received.

Data exchange is accomplished by directly sending messages containing a write or a read request. While the initiating process is waiting for a response, the simulation time can proceed until the simulated round-trip time is reached. If there is still enough local work available, the speed of the virtual prototype will not be diminished. In case, the read data is mandatory for continuing the local simulation, the SystemC instance has to be paused until the response was received. This is depicted in Fig. 2 at the right-hand side.

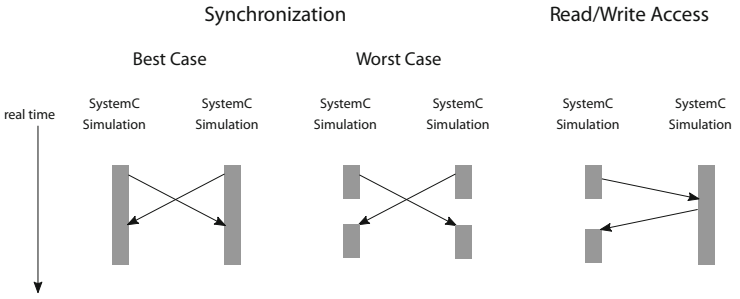

Fig. 2. Best and worst case of the presented approach for time synchronization and a data exchange example between two SystemC instances. Gray boxes are show when the simulation on the machine progresses. The arrows depict messages.

3.4 Peripherals

Components like memory are often provided by frameworks like OVP and gem5. However, accessing the data is only possible from within these simulators which makes usage from the outside difficult. As a consequence, the necessary memory and the input and output devices had to be implemented as reusable SystemC modules. This allows access of the GPU and realtime CPU models with their specific requirements like the need to directly access the data using pointers. After the creation of the virtual prototype, an evaluation of possible bottlenecks was done. The following Section gives an insight into the hindrances of the given approach.

4 Evaluation

All of the presented simulation models are already available as standalone versions. However, connecting them to a SystemC runtime causes speed impacts by making certain optimization methods like just-in-time compilation difficult or even impossible. Figure 3 shows the data paths that are analyzed in this Section. Section 4.1 covers the overhead introduced by using the SystemC connectors of the mentioned general purpose CPU simulators. This corresponds to data path (1) within the Figure. (2) belongs to the overhead of the newly written CUDA connector module and the data exchange between the CUDA runtime library and a test memory which is measured in Sect. 4.2. Data path (3) of the module created for the distributed SystemC simulation is evaluated in Sect. 4.3. Its messages are exchanged with another SystemC instance which can be located on the same computer or on another computer.

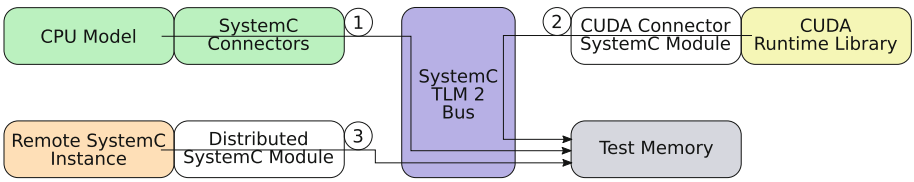

Fig. 3. The analyzed impacts. Each arrow represents one of the three analyzed data paths. The white, purple, and gray boxes are modules that were implemented for this paper. (Color figure online)

4.1 General Purpose CPU

To evaluate the impact of modeling the bus and memory system with the means of SystemC instead of the native possibilities of gem5 and OVP, two virtual prototype designs were created for each model. For gem5, the first design includes the *SimpleMemory* module as main memory. The second one uses the presented bus and memory system for heterogeneous simulation. Similar to this, the first

design of OVP uses its native main memory while the second variant uses the presented memory layout. The ARM Fast Models use the SystemC Modeling Library, which is developed by Synopsys and compatible to TLM 2, to connect to the memory. Since there is no native way to provide a memory implementation, the SystemC overhead could not be analyzed in an isolated way.

As reference benchmarks CoreMark[2], an implementation of the Ackermann function, a Monte Carlo algorithm for calculating Pi, and the Sieve of Eratosthenes were used. These programs are expected to represent different kinds of real world problems that could be run on a general purpose processor. Table 1 shows the slowdown experienced for each benchmark from the native use of peripherals in comparison to the SystemC versions.

Table 1. The overhead introduced by coupling the simulators with SystemC. A value of one means no overhead while a value of two means that twice the time is required.

	CoreMark	Ackermann	Monte Carlo	Sieve
gem5	2.7	3.0	3.1	3.1
OVP	798	377	284	1291

Gem5's slowdown ranges from 2.7 to 3.1 which means that the time required to run one of the programs with SystemC is approximately three times as long as the native implementation. An investigation about the cause of this slowdown using the SystemC version showed that around 43.8% of the simulation time was spent in the runtime and peripheral code. Additionally, marshalling and unmarshalling packages from gem5 to and from TLM takes some time. This in combination with memory allocations and memory copy operations is accountable for another 19.3% of the time. Only 32.7% of the time is actually used for simulating the processor. The remaining 4.2% are spent in various C or C++ runtime functions.

OVP suffers a much larger slowdown due to lost optimization potentials when using the SystemC coupling for the main memory. The code morphing (OVP's name for Just-In-Time compilation) cannot deliver enough speedup any more because OVP cannot assume that the instructions stay the same. Thus, it has to fetch them every time anew always suffering a round-trip time to the SystemC memory implementation and back. In total, 85% of the simulation time is spent in the SystemC part of the virtual platform.

As shown in this Section, the simulation performance of the general purpose simulators is tremendously diminished when the SystemC binding is used. This is caused by the overhead introduced by converting the data requests from the internal representation to a TLM compatible one. Additionally, no features of TLM are used which would allow a speedup again. For example, the DMI can be

[2] https://www.eembc.org/coremark (accessed on 2018-12-04).

used to obtain a pointer into the memory which avoids a lot of overhead which was measured in this Section. Hence, some optimizations should be implemented to increase simulation speed.

4.2 GPU

The SystemC module for linking against GPU Ocelot and GPGPU-Sim does not introduce relevant overhead. This was evaluated by measuring the time of the CUDA simulations once without the SystemC connection as the libraries are intended to be used and once with a CPU model and the SystemC connection in place. To get only the impact on the accelerator code without interference from the required host code, the CUDA runtime library source code was modified to cumulate the time used within the CUDA runtime functions. Multiple different algorithms were run to even out software specific anomalies. The benchmarking applications include a vector addition (vecAdd) which was done for a vector containing one million elements, ten matrix multiplications (matrixMult) of 320 × 320 and 320 × 640 matrices, 128 iterations of the Black Scholes algorithm [4] with a problem size of 5000, and a sobel algorithm which is sometimes used as a component of an ADAS application, e.g. in lane detection algorithms [14]. From a set of at least ten measurements always the fastest results were used and the overhead determined. It is shown in Table 2 for all four algorithms. The Host Runtime corresponds to the time measured without any SystemC involvement while the simulation runtime (Sim. Runtime) corresponds to the time measured with the CUDA library connected to the virtual prototype.

Table 2. Overhead introduced by the SystemC connection module in comparison to native usage of the CUDA simulation libraries for different benchmark algorithms.

	vecAdd	matrixMult	Black Scholes	Sobel
Overhead	3.7%	0.5%	1.3%	2.0%
Host Runtime	6.6 s	587.6 s	13.3 s	23.4 s
Sim. Runtime	6.8 s	590.7 s	13.5 s	23.9 s

As can be seen from Table 2 the overhead is relatively small and stays below 4% for all investigated benchmarks. Especially long running algorithms like the matrix multiplication are hardly affected by the SystemC module. Short running ones like the vector addition display a bigger overhead which is still small in comparison to the overhead introduced to the general purpose CPU models for example. The source of the overhead lies within the SystemC connector that has to copy the operands from the virtual prototype to the CUDA runtime library and is responsible for performing the address translations. Since the remaining work which contains the work-intensive tasks like the kernel code is executed separately from the virtual prototype, the impact is kept low. Hence, the longer a kernel runs the less overhead is experienced.

4.3 Realtime Processor

Since the AURIX model uses a network connection to connect to the rest of the simulation, the impacts of this code on the system was investigated. To determine the overhead introduced by the proposed approach, synthetic benchmarks were created. They consist of a worker thread that has to be dispatched once each simulated nanosecond meaning a frequency of 1 GHz. It was run first without any networking code to obtain a reference runtime that can be compared. Each measurement was done at least ten times and the average of all runs was taken to minimize the impacts from the host operating system on the results.

At first, only the overhead introduced by the periodic synchronization events was determined. For this, different times between sending the synchronization messages were considered. A period interval of one nanosecond means that the worker thread and the synchronization thread are run alternately. A period interval of two nanoseconds means that for two runs of the worker thread body, one run of the synchronization thread occurs. Figure 4 shows the relative runtime the synchronization messages introduce on the worker thread. A value of zero represents no overhead while a value of one implies a runtime that takes twice as long as the local reference. The measurements were done with two different computers connected via an Ethernet switch and locally on one host by using the loopback device. Additionally, the standard deviation for the measurements was calculated.

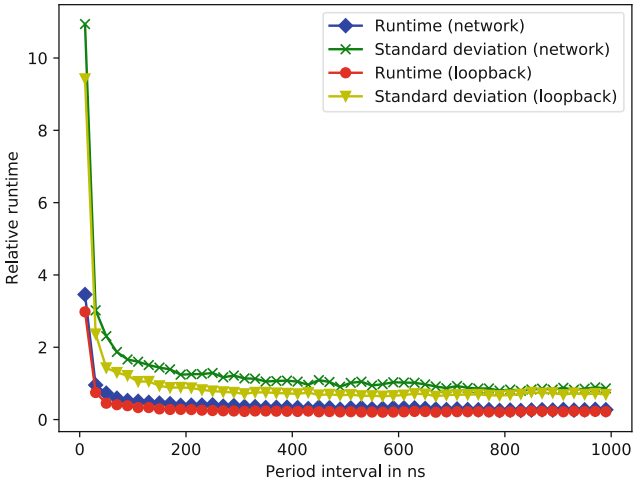

Fig. 4. The relative runtime and its standard deviation with the synchronization messages enabled in comparison to the local reference time once done over network and once using the local loopback device.

As can be seen, a period interval of 1000 ns reduces the overhead to 20–25%. This means that having an interval length that is 1000 times longer than the default clock rate of the system should reduce the impact from more than 300% in case every nanosecond a message is sent to only 20–25%. A similar shape can be seen in Fig. 5 which shows the overhead depending on the allowed simulation time discrepancy between the two SystemC instances. The period was fixed to 1000 ns to reduce the overhead introduced by the periodic sending operation. With an allowed discrepancy of about 8000 ns, the measurable overhead is nearly the same as with only sending the synchronization messages: A little bit above 25%. This should be the time of the best case presented in Fig. 2. It is noticeable that the major impact on the overhead introduced by the synchronization mechanism is depending on the selected period (1000 ns) since the overhead gets reduced at steps of 1000 ns of allowed discrepancy. This is due to the fact that each instance waits for the synchronization message while it is not sent yet. It can be concluded that the allowed discrepancy should be approximately eight times the period time to reduce the overhead.

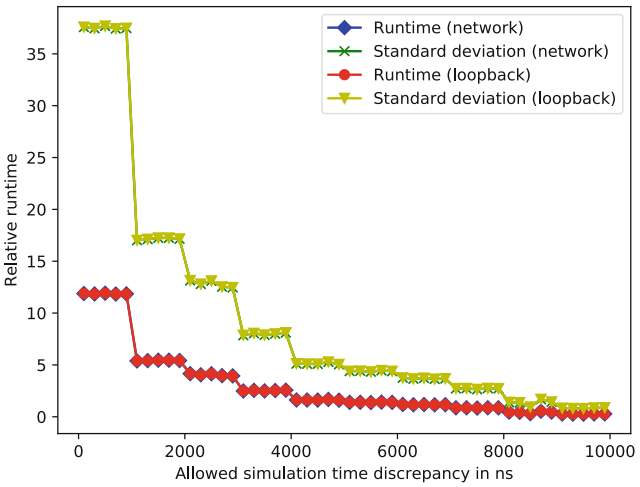

Fig. 5. The relative runtime and its standard deviation with depending on the allowed simulation time discrepancy in comparison to the local reference time once done over network and once using the local loopback device.

Finally, the overhead when sending TLM packages via the network was analysed. The period was fixed to 1000 ns and the discrepancy to 8000 ns. Since the overhead introduced is directly depending on the SystemC design and a generic result cannot be given, the indirect overhead of another TLM data exchange on the synthetic worker was measured. Thus, another thread was introduced that sends as much data as the latency allows. Figure 6 shows that the complete overhead via network is around 50% even for the smallest and greatest evaluated latencies. As a consequence, no real advice can be given regarding the best

suitable latency. The best case would be if the latencies between the remote and the local simulator instances can be set equal to the latencies of the real hardware. When using the loopback device, the overall overhead can be reduced to approximately 30%. However, this cannot be done for the presented virtual prototype due to the requirement of different host computers.

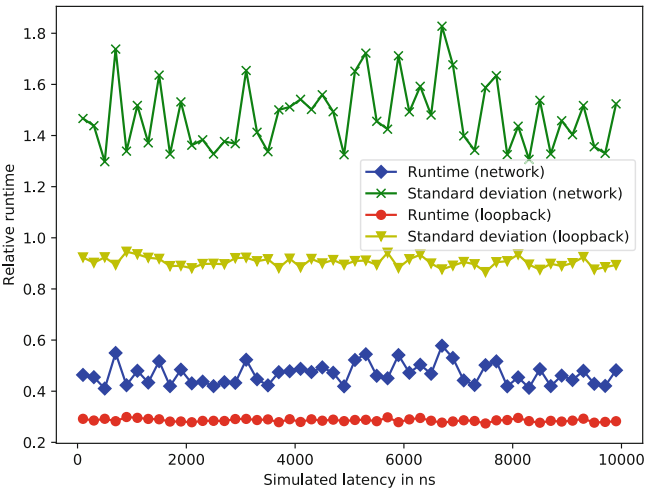

Fig. 6. The overhead and its standard deviation introduced by another thread sending TLM messages using the presented network approach. This was once done via network and once via the loopback device.

5 Future Work

From the analysis, the speed can be identified as a major issue in regard to the usability of the system. While evaluation of small workloads on the heterogeneous system can be feasible, larger sensor processing algorithms (e.g. working on camera pictures) will take too long for a functional run on the simulated platform. Hence, certain optimization steps from within the involved processor models should be implemented. One simple improvement can be the usage of DMI as already stated above. Additionally, assertions should be given to allow the complete exploitation of Just-In-Time techniques. For example, direct changes of the underlying SystemC memory that may also contain instructions should be forbidden. Callback functions may then be used to invalidate the memory (like done for the DMI) if it is changed.

From the findings of this paper, other connection approaches without unconditional compatibility might also achieve higher speeds. Since the isolated way of execution of the CUDA simulation achieves the best speed, it seems beneficial to also isolate the general purpose CPUs. However, this comes with its own

additional challenges like how to realize direct pointers into the host memory which are required by the GPU emulation.

Further improvements can be expected by using models or enabling features in the selected models that determine the runtime and power behavior of the real hardware when the simulated software is run on it. While this is supported by gem5 and GPGPU-Sim to a certain degree, there are still deviations from the reference hardware. Additionally, the whole bus system has to be modelled accurately which is difficult without further insight into today's ADAS platforms. These enhancements could lead to a virtual prototype allowing a very detailed evaluation of a heterogeneous system as it might be required for certification.

6 Conclusion

In this paper, an approach for functionally simulating a heterogeneous system using already available processor models was shown. SystemC was used as a common communication language and additional modules for connecting the CUDA GPU simulator, and a remote connection to realtime processors were created. In comparison to standalone simulation, severe performance penalties were noticed. As bottlenecks, no longer functioning performance optimizations of the general purpose CPU emulators were identified slowing down the simulation by a factor between 2.7 (best case with gem5) up to a factor of 1291 (worst case with OVP). Additionally, the overhead introduced by the remote connection used to communicate with the realtime processor was analyzed. It could be shown that it stays below 65% for the synthetic benchmarks. For the GPU binding, a very small simulation runtime impact could be observed that stayed below 4% for the observed benchmark applications.

References

1. Aaamodt, T., Boktor, A.: GPGPU-Sim 3.x: a performance simulator for many-core accelerator research. In: International Symposium on Computer Architecture (ISCA) (2012). http://www.gpgpu-sim.org/isca2012-tutorial
2. Anwar Taie, M.: New trends in automotive software design for the challenges of active safety and autonomous vehicles. In: FAST-zero'15: 3rd International Symposium on Future Active Safety Technology Toward Zero Traffic Accidents 2015 (2015)
3. Binkert, N., et al.: The gem5 simulator. SIGARCH Comput. Archit. News **39**(2), 1–7 (2011)
4. Black, F., Scholes, M.: The pricing of options and corporate liabilities. J. Polit. Econ. **81**(3), 637–654 (1973)
5. Delicia, G.S.P., Bruckschloegl, T., Figuli, P., Tradowsky, C., Almeida, G.M., Becker, J.: Bringing accuracy to open virtual platforms (OVP): a safari from high-level tools to low-level microarchitectures. In: IJCA Proceedings on International Conference on Innovations in Intelligent Instrumentation, Optimization and Electrical Sciences ICIIIOES, no. 10, pp. 22–27. Citeseer (2013)

6. Diamos, G.F., Kerr, A.R., Yalamanchili, S., Clark, N.: Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, PACT 2010, pp. 353–364. ACM, New York (2010)
7. Dikmen, M., Burns, C.: Trust in autonomous vehicles: the case of tesla autopilot and summon. In: 2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC), pp. 1093–1098, October 2017
8. Greenblatt, N.A.: Self-driving cars and the law. IEEE Spectr. **53**(2), 46–51 (2016)
9. Gutierrez, A., et al.: Lost in abstraction: pitfalls of analyzing GPUs at the intermediate language level. In: 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 608–619, February 2018
10. IEEE Computer Society: IEEE Standard for Standard SystemC Language Reference Manual. IEEE Std 1666–2011 (2012)
11. Leupers, R., et al.: Virtual platforms: breaking new grounds. In: 2012 Design, Automation Test in Europe Conference Exhibition (DATE), pp. 685–690, March 2012
12. Menard, C., Jung, M., Castrillon, J., Wehn, N.: System simulation with gem5 and Systemc: the keystone for full interoperability. In: Proceedings of the IEEE International Conference on Embedded Computer Systems Architectures Modeling and Simulation (SAMOS). IEEE, July 2017
13. Power, J., Hestness, J., Orr, M.S., Hill, M.D., Wood, D.A.: gem5-gpu: a heterogeneous CPU-GPU simulator. IEEE Comput. Archit. Lett. **14**(1), 34–36 (2015)
14. Reichenbach, M., Liebischer, L., Vaas, S., Fey, D.: Comparison of lane detection algorithms for ADAS using embedded hardware architectures. In: 2018 Conference on Design and Architectures for Signal and Image Processing (DASIP), pp. 48–53, October 2018
15. Schoenwetter, D., Ditter, A., Aizinger, V., Reuter, B., Fey, D.: Cache aware instruction accurate simulation of a 3-D coastal ocean model on low power hardware. In: 2016 6th International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH), pp. 1–9, July 2016
16. Skende, A.: Introducing "parker": next-generation tegra system-on-chip. In: 2016 IEEE Hot Chips 28 Symposium (HCS), August 2016
17. Ubal, R., Jang, B., Mistry, P., Schaa, D., Kaeli, D.: Multi2Sim: a simulation framework for CPU-GPU computing. In: 2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 335–344, September 2012
18. Weinstock, J.H., Leupers, R., Ascheid, G., Petras, D., Hoffmann, A.: Systemc-link: parallel systemc simulation using time-decoupled segments. In: 2016 Design, Automation Test in Europe Conference Exhibition (DATE), pp. 493–498, March 2016