Fast SystemC Processor Models with Unicorn

Lukas Jünger
RWTH Aachen University
juenger@ice.rwth-aachen.de

Jan Henrik Weinstock
RWTH Aachen University
weinstock@ice.rwth-aachen.de

Rainer Leupers
RWTH Aachen University
leupers@ice.rwth-aachen.de

Gerd Ascheid
RWTH Aachen University
ascheid@ice.rwth-aachen.de

ABSTRACT

In this work a Virtual Platform (VP) is presented containing a novel processor model for the latest ARMv8 instruction set architecture. This processor model was constructed using the Unicorn emulator [6]. The necessary modifications to the Unicorn emulator and subsequent performance improvements during SystemC simulation are shown in detail. In addition the integration into a VP using a state-of-the-art SystemC modeling library is described. A comparison is made with a VP containing another similar processor model, highlighting the benefits of using Unicorn for processor modeling in a SystemC environment.

CCS CONCEPTS

• **Hardware** → **Hardware-software codesign**; • **Computing methodologies** → *Discrete-event simulation*;

KEYWORDS

Electronic System Level, QEMU, SystemC

ACM Reference Format:
Lukas Jünger, Jan Henrik Weinstock, Rainer Leupers, and Gerd Ascheid. 2019. Fast SystemC Processor Models with Unicorn. In *Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO '19), January 21–23, 2019, Valencia, Spain.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3300189.3300191

1 INTRODUCTION

In the fast-paced development cycles of today, Virtual Platforms (VPs) have become an essential tool in embedded software development. VPs are typically available long before their hardware counterparts, allowing to start software development earlier. They enable hardware/software co-design, since insights gained from early software development can be of use to the hardware designers. Even though VPs do not model every detail of real hardware, having more time for software development and testing yields more mature and stable software. This, in turn, speeds up the bringup on the physical hardware of the final product once it is available.

RAPIDO '19, January 21–23, 2019, Valencia, Spain

ACM ISBN 978-1-4503-6260-3/19/01…$15.00
https://doi.org/10.1145/3300189.3300191

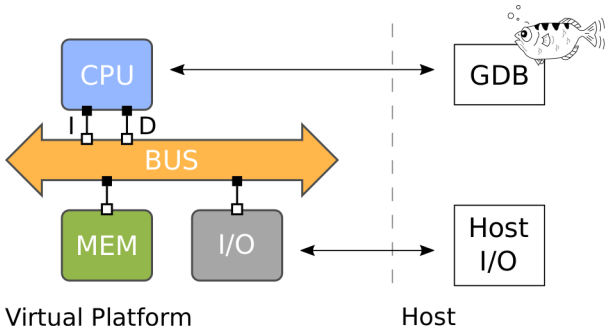

Figure 1: Overview of example VP

SystemC Transaction-Level Modeling 2.0 [5] (TLM) has become the de facto standard for constructing VPs. It allows for the VP to be described using C++, which is then compiled into a fast simulation executable. Because the VP is a C++ program, it can be inspected using off-the-shelf C++ debuggers and other analysis tools. Hence, VPs offer a high degree of introspection, making them ideal for debugging, and their malleability and interoperability allows users to customize them to their needs. An overview of an exemplary VP intended for software development is given in Figure 1. It consists of a processor model with debugger integration, memory and an Input/Output (I/O) device, which is connected to host I/O.

Ideally, there should be no discernible difference between using a VP or physical hardware from the point-of-view of the software developer. Notably, it should react to inputs as quickly as physical hardware. Therefore, rapid simulation is a key productivity feature of a VP. A VP usually consists of multiple component models such as processors, buses, memories and peripherals. Since all of these components are simulated, they have an effect on the overall simulation performance. The processor model is a core component of the VP and normally contains an Instruction Set Simulator (ISS) for the simulated target Instruction Set Architecture (ISA). During simulation a substantial amount of execution time is spent in the ISS of the processor model. Therefore, a high performance ISS is paramount to achieving rapid simulation speed, yielding smooth VP operation in conventional software development environments.

In this work, a TLM processor model using the open-source Unicorn emulator [6] is introduced. ARMv8 was selected as the target ISA, because it is the most common 64-bit embedded systems ISA. This processor model was subsequently used to construct a full VP. Different benchmarks were evaluated on the VP to assess the performance of the Unicorn-based processor model.

The remainder of this work is structured as follows. In Section 2, work related to the topic is surveyed and summarized. Afterwards, Section 3 describes the Unicorn TLM wrapper and GNU Debugger (GDB) integration. The VP, that was built using the new Unicorn-based processor model, is described in Section 4. An experimental evaluation of the VP with industry-standard benchmarks is given in Section 5. Finally, a conclusion is drawn in Section 6.

The contributions of this work are:

- Design of a novel ARMv8 SystemC processor model using the Unicorn emulator
- A new instruction counting mechanism for the Unicorn emulator, suitable for rapid SystemC simulation
- Construction of a realistic VP using the novel processor model
- Assessment of the performance of the Unicorn-based VP in comparison with a similar state-of-the-art VP

2 RELATED WORK

Generally, an ISS is needed when building a processor model from scratch. ARM offers commercial processor models, that are equipped with a SystemC interface: the ARM FastModels [1]. Besides the SystemC interface, these models offer different debug and scripting Application Programming Interfaces (APIs).

Another option is the open-source QEMU system emulator [2]. QEMU contains a Dynamic Binary Translation (DBT) ISS component for many different ISAs.

The main idea of DBT is to translate binary instructions from the target to the host instruction set at runtime. QEMU also stores and then re-uses the translated instructions. This way, the target instruction fetch and decode is only executed once as opposed to an interpreting ISS. Different optimization techniques can be applied to speed up the DBT process. QEMU is not a stand-alone processor model. It also includes models of other hardware components, including, but not limited to, graphics card, sound card, USB controllers and memory controllers. By combining these components, QEMU is capable of simulating a complete system. In addition, it includes a GDB server, which allows the user to connect a GDB instance to debug the software running on the simulated system. However, QEMU does not have a SystemC interface and is geared towards full system emulation. It is not usable in a VP by itself, since it lacks the customizability and interoperability of SystemC.

To address the lack of a SystemC interface in QEMU, the QBOX platform emulation environment by GreenSocs [3] adds a SystemC wrapper to the emulator. With this, QEMU can be integrated into a TLM simulation. It also keeps the GDB server intact to enable debugging of the target software. Since QEMU is kept mostly unmodified, including the hardware component models, there is inherent overhead if only a processor model is needed in the VP. AMVP [9], selected as comparison in this work, uses QBOX in its processor model.

In 2015 Nguyen et al. presented their Unicorn emulator [6]. Unicorn is based on the QEMU system emulator version 2.2.1, but it keeps only a small subset of the functionality of QEMU. The fast DBT based ISSs are kept, while the other component models are removed. A lightweight API is provided to use the ISSs. Unicorn does not include a TLM compatible interface and thus cannot be used in a SystemC VP without modification. This work proposes a solution to this problem.

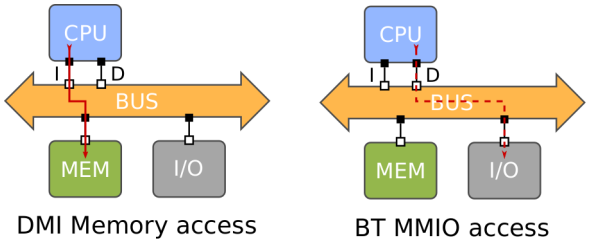

Figure 2: Different kinds of memory accesses

3 EMBEDDING UNICORN IN SYSTEMC

In this work, the Unicorn emulator [6] is used to build a TLM-compatible ARMv8 processor model. Since Unicorn does not include a TLM interface, some modifications are made to the existing code base. In this section, these modifications are described in more detail. The Unicorn-based processor model was constructed using the Virtual Components Modeling Library (VCML) [10]. VCML provides base classes, that simplify the construction of SystemC hardware components models. It also includes several complete SystemC models for assembling a VP, such as memory, bus and peripheral models.

From the perspective of VCML, Unicorn is a processor model. As such, mechanisms for memory access, Memory-Mapped I/O (MMIO) and interrupts had to be designed. These are described in Section 3.1. In addition, VCML includes a GDB Remote Serial Protocol (RSP) [7] server implementation, that simplifies the debugger integration, which is essential for a VP. Without it, the VP lacks core features for software debugging and development. Therefore, GDB support was added in the Unicorn-based processor model. More details on the GDB integration are provided in Section 3.3. Unicorn is not designed for speed and thus specific performance enhancements were added. These enhancements are described in Section 3.4.

3.1 Memory

For this work, two types of memory accesses have to be distinguished. The first type is a standard memory access from the processor model to the RAM model. The second type of memory access is MMIO, which happens when the processor model accesses a memory-mapped peripheral e.g. a UART. In the SystemC domain, these two kinds of accesses have to trigger different behaviors. When the processor model is accessing the RAM, the access can be conducted using the TLM Direct Memory Interface (DMI). In this case the memory access is done directly via a pointer, which is the fastest method for modeling this kind of behavior. In case the processor model is accessing a MMIO peripheral model, the memory access has to be conducted via the TLM Blocking Transport interface (BT). This is due to the fact, that the peripheral model usually has to react to the access, e.g., by printing a character on the screen, and thus needs to be informed of it. The two different types of memory accesses are depicted in Figure 2.

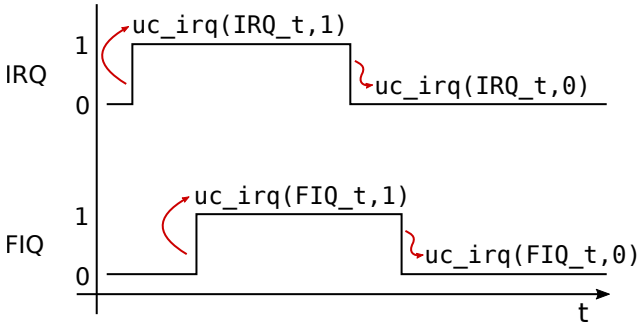

Figure 3: External interrupts triggering function calls to the processor model

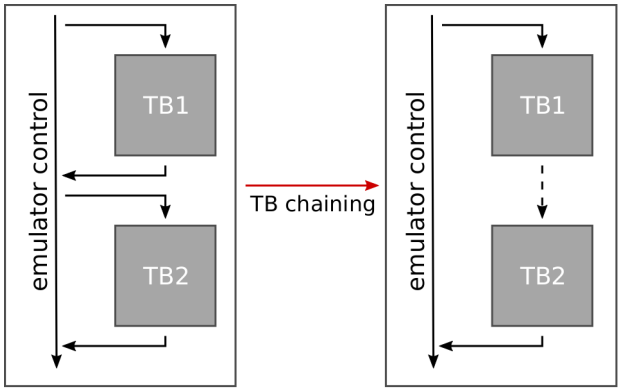

Figure 4: Execution flow before and after TB chaining optimization

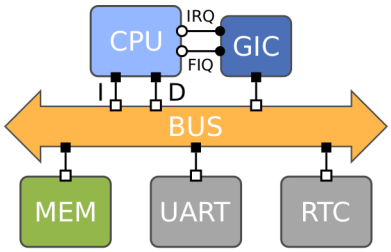

Figure 5: Overview of the Unicorn VP

To the processor it is generally unknown whether a memory access targets the RAM or a MMIO peripheral. The distinction between the different memory access methods is therefore to be made by the TLM wrapper. The processor fetches instructions and data from the RAM via its corresponding data and instruction ports, which are connected to a bus model. Unicorn includes a mechanism that allows for mapping host memory directly to the emulator memory space via a pointer. This mechanism is used for mapping the RAM model memory to the emulator at the end of the SystemC elaboration phase. After the RAM is mapped, program execution can start.

Unicorn allows to connect callback functions to different memory events, such as an unmapped memory access. This callback function is executed when an instruction accesses memory that is not mapped in the address space of the emulator. When this happens, the emulator cannot continue execution and the following algorithm is executed to resolve the issue:

(1) Try to acquire a DMI pointer and map the missing memory into the memory space of the emulator.
(2) If a DMI pointer cannot be acquired, a BT is used for the memory access, and callback functions are registered on the address to handle the memory access via a BT in the future.

Using the method described above, all memory accesses occurring during the SystemC simulation can be handled.

3.2 Interrupts

ARMv8 processors feature the external IRQ and FIQ interrupt signals, which have to be included in the processor model to achieve full compatibility The processor model has to be able to react to changes in these signals. Unfortunately, the Unicorn API does not expose the interrupt functionality of the ISS. Therefore, the API had to be extended to expose this functionality to the TLM wrapper. Interrupts are modeled as `sc_signal<bool>`. The processor model reacts the signal changes by executing an IRQ trigger (`uc_irq()`) function on rising and falling signal edges. This is depicted in Figure 3.

In this IRQ trigger function, the corresponding QEMU function is called via the extended Unicorn API, initiating the CPU state changes to reflect the occurrence of the interrupt.

3.3 GNU Debugger Integration

In order to ease the bringup of the target software on the processor model, a debugger integration is indispensable. The VCML processor model base class includes an integration of GDB RSP. GDB can be connected to the running VP remotely via a network connection and does not need to be executed on the same machine.

For GDB support, the processor model needs to be able to read and write the emulated processor registers and set and remove breakpoints. Many of the ARMv8 registers are accessible via the Unicorn API and the missing ones were added for this work. Breakpoint support was added using Unicorn memory callback functions. When GDB sets a breakpoint on the processor model, a memory callback is added for the corresponding memory address. Once execution reaches this address, the VCML GDB server stops execution and passes control to GDB to resume interactive debugging. Memory access is provided to GDB by the VCML GDB integration directly via the TLM data and instruction sockets, and therefore needs not to be implemented by the processor model.

3.4 Performance Optimization

TLM simulations use temporal decoupling to increase simulation performance. Following this concept, the processor model is allowed to run ahead of the global simulation time until the next

synchronization point is reached. The largest amount of time a thread may differ from the global simulation time is referred to as the *quantum*. The size of the quantum corresponds to the amount of instructions the processor model may execute in between two synchronization points. In order to be able to execute a certain number of instructions, there needs to be an instruction counter and a way to exit the emulator when the desired number of instructions was executed. There are several ways in which the instruction counting can be implemented. In order to evaluate the different methods, more detailed knowledge of the Unicorn emulator is required.

As mentioned above Unicorn uses the DBT ISSs from the QEMU system emulator. The main component of these ISSs is the Tiny Code Generator (TCG), which handles the DBT. First, the target ARMv8 instructions are translated into TCG instructions, which are then optimized and translated to host instructions. The TCG translates on Translation Block (TB) granularity. A TB is a coherent block of instructions that can only be entered at its beginning and exited at its end, e.g., via a branch instruction. To further improve performance the TCG has a TB cache, that stores previously translated TBs for later reuse. At the end of a TB, execution can continue in two places, which correspond to whether the branch at the end of the TB was taken or not. This is used for an optimization referred to as *TB chaining*. When the end of a TB is reached, Unicorn checks which TB to execute next. It then stores the information on which TB followed due to which branch decision. When the TB is executed next time and the same branch decision is made, the execution directly continues at the correct TB without the need for another TB lookup. If at this point the other branch option is taken, this information is also stored together with the subsequent TB and when the TB is executed again, execution can continue directly for either branch decision. This optimization is depicted in Figure 4.

Unicorn implements instruction counting by branching to a counting routine after every instruction. This counting routine also checks whether execution should stop, in case the desired number of instructions was executed. Since Unicorn allows stopping execution at any address, the TB cache has to be flushed regularly. This is necessary, because Unicorn can also stop and continue execution somewhere inside a TB and not only at the beginning and end. In this case, at least the corresponding TB has to be re-translated. However, Unicorns default behavior is to flush the entire TB cache. When a precise instruction count is not needed, this behavior introduces considerable performance overhead, since it effectively degrades the TCG to an interpreter. In a TLM simulation a small TLM quantum overshoot is acceptable, meaning that execution can always continue until the end of the TB is reached. Therefore, it is sufficient to only count instructions on the TB level. This can be done efficiently by adding host machine instructions at the beginning of the TB to increase an instruction counter in host memory by the number of instructions in this TB. This also works when TBs are chained. Branching to a counting routine is not necessary. For this work, an instruction counting mechanism per TB, as described above, was added to the Unicorn emulator. The performance difference between the two methods is shown in Section 5.

The default behavior of Unicorn is still useful for debugging, where single instruction steps are common. Thus, the default behavior is enabled in the processor model when the debugger issues a single instructions step. When the debugger issues a continue command, the faster instruction counting mode is re-enabled.

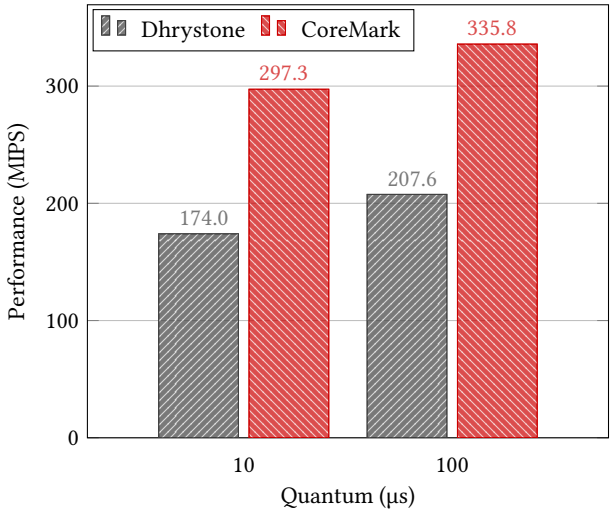

Figure 6: Benchmark results for Dhrystone and CoreMark on the Unicorn VP

4 THE UNICORN VIRTUAL PLATFORM

The Unicorn-based SystemC processor model, described in Section 3, was used to build a VP for experimental evaluation. This VP is described in this section. Figure 5 shows a platform overview of the complete VP.

Besides the processor model, the VP contains the following components:

- A generic bus model
- An ARM GICv2 interrupt controller model
- A generic memory model
- A Real-Time Clock (RTC) model
- A PL011 ARM PrimeCell UART peripheral model

All components besides the RTC are part of the VCML [10]. The RTC was implemented for this work. It is used to provide a reference time to the benchmark target software. The target software is copied to the memory before the SystemC simulation starts.

5 EXPERIMENTAL EVALUATION

To evaluate the performance of the processor model, described in Section 3, different benchmarks were executed on the Unicorn VP, introduced in Section 4. CoreMark [4] and Dhrystone [8] were ported to the VP as industry-standard benchmarks. The following results were acquired on an octa-core Intel i7-7700 host system with 32GB RAM, running CentOS 7.3.1611. Each experiment was repeated ten times and the results were averaged for the final result.

First, the CoreMark and Dhrystone benchmarks were executed on the Unicorn VP with set SystemC quanta of 10 µs and 100 µs. The processor model performance was measured in Million simulated Instructions Per wall clock Second (MIPS) using the profiling facilities of the VCML. The results are summarized in Figure 6.

It can be observed, that the CoreMark benchmark executes faster than the Dhrystone benchmark. This is likely caused by the fact, that Dhrystone is more memory intensive and the processor model

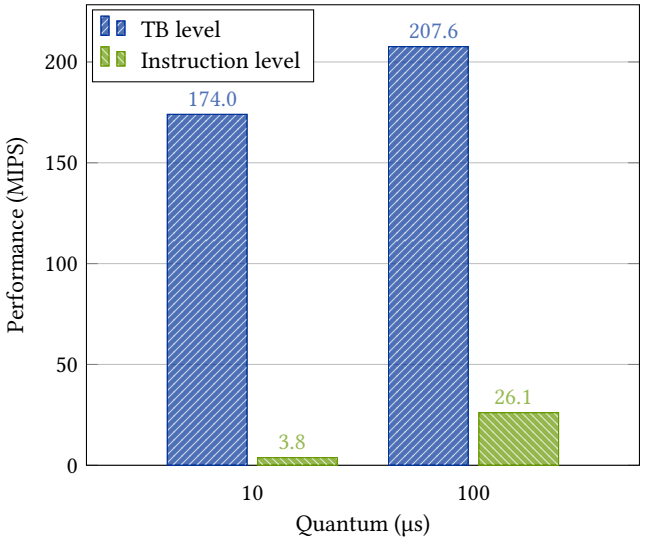

Figure 7: Benchmark results for Dhrystone on the Unicorn VP with TB and instruction granularity counting

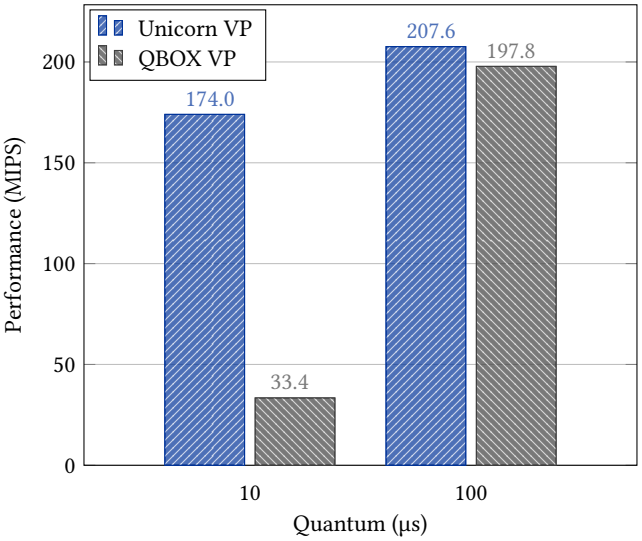

Figure 8: Benchmark results for Dhrystone on the QBOX and Unicorn VPs

has to leave the ISS more frequently to access the memory, which is costly.

As described in Section 3.4, the performance of Unicorn in a SystemC VP was improved by adding a novel instruction counting mechanism. The performance improvement was experimentally evaluated with the Dhrystone benchmark, which was executed on the Unicorn VP with the proposed TB-granularity and the original instruction granularity counting mechanism. The MIPS performance of the processor model was measured using the VCML profiling facilities. The results are visualized in Figure 7. Here it can be observed, that a significant performance improvement was achieved. However, one has to keep in mind that this comes at the cost of accuracy, since now TLM quantum overshoots must be tolerated. However, in a loosely-timed simulation this is generally not a problem.

In order to assess the processor model performance in comparison with similar models, a comparison with a single-core, sequential version of the QBOX-based [3] AMVP [9] was performed. The Dhrystone benchmark was executed on both VPs with two different TLM quanta of 10 µs and 100 µs. As before, the MIPS performance was measured using the VCML profiling facilities in both VPs. The results are summarized in Figure 8.

The Unicorn VP outperformed AMVP in both settings. It can be observed, that Unicorn was significantly faster than AMVP for the 10 µs quantum. From this it can be deduced, that the overhead of entering and leaving the QBOX emulator dominates the performance difference. For the 10 µs quantum the ISS has to be paused and resumed more often, which is why the lightweight Unicorn-based processor model is significantly faster here. With a higher quantum, more time is spent inside the ISS. Since Unicorn and QBOX both use the QEMU TCG, the performance difference is not as significant.

6 CONCLUSION

This work shows, that Unicorn is a promising candidate for use as a TLM processor model. The missing TLM wrapper can be implemented using Unicorn callback functions and minor extensions to the Unicorn API. A big advantage is, that Unicorn is lightweight and only includes what is absolutely necessary for processor emulation. It keeps the QEMU DBT ISS with TCG and with some modifications can be brought to rapid simulation speed. The performance of the constructed VP is sufficient for interactive debug and higher than comparable VPs. When the VP is used as a debug target, no difference is observable between the VP and physical hardware.

REFERENCES

[1] ARM Holdings. Virtual Prototypes: Fast Models. https://developer.arm.com/products/system-design/fast-models. Online; accessed 21-10-2018.
[2] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, 2005.
[3] G. Delbergue, M. Burton, F. Konrad, B. Le Gal, and C. Jego. QBox: an industrial solution for virtual platform simulation using QEMU and SystemC TLM-2.0. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, 2016.
[4] Embedded Microprocessor Benchmark Consortium. CoreMark: An EEMBC Benchmark. http://www.eembc.org/coremark, 2018. Online; accessed 21-10-2018.
[5] IEEE. IEEE Standard for Standard SystemC Language Reference Manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, Jan 2012.
[6] A. Q. Nguyen and H. V. Dang. Unicorn: Next Generation CPU Emulator Framework. In *Proceedings of the 2015 Blackhat USA conference*, 2015.
[7] R. Stallman, R. Pesch, and S. Shebs. GDB Remote Serial Protocol. In *Debugging with GDB: the GNU Source-Level debugger, Version 5.1.1*, pages 281–292. Free Software Foundation, Boston, USA, 9 edition, 2002.
[8] R. P. Weicker. Dhrystone Benchmark: Rationale for Version 2 and Measurement Rules. *SIGPLAN Not.*, 23(8):49–62, Aug. 1988.
[9] J. H. Weinstock, R. L. Bücs, F. Walbroel, R. Leupers, and G. Ascheid. AMVP - A High Performance Virtual Platform Using Parallel SystemC for Multicore ARM Architectures: Work-in-progress. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, CODES '18, pages 13:1–13:2, Piscataway, NJ, USA, 2018. IEEE Press.
[10] Weinstock, Jan Henrik. Virtual Components Modeling Library. https://github.com/janweinstock/vcml. Online; accessed 21-10-2018.