ARM-on-ARM: Leveraging Virtualization Extensions for Fast Virtual Platforms

Lukas Jünger *, Jan Luca Malte Bölke †, Stephan Tobies †, Rainer Leupers *, and Andreas Hoffmann †
*RWTH Aachen University, {juenger,leupers}@ice.rwth-aachen.de
†Synopsys GmbH, {jan.bolke, stephan.tobies, andreas.hoffmann}@synopsys.com

***Abstract*—Virtual Platforms (VPs) are an essential enabling technology in the System-on-a-Chip (SoC) development cycle. They are used for early software development and hardware/software codesign. However, since virtual prototyping is limited by simulation performance, improving the simulation speed of VPs has been an active research topic for years. Different strategies have been proposed, such as fast instruction set simulation using Dynamic Binary Translation (DBT). But even fast simulators do not reach native execution speed. They do however allow executing rich Operating System (OS) kernels, which is typically infeasible when another OS is already running.**

Executing multiple OSs on shared physical hardware is typically accomplished by using virtualization, which has a long history on x86 hardware. It enables encapsulated, native code execution on the host processor and has been extensively used in data centers, where many users share hardware resources. When it comes to embedded systems, virtualization has been made available recently. For ARM processors, virtualization was introduced with the ARM Virtualization Extensions for the ARMv7 architecture. Since virtualization allows native guest code execution, near-native execution speeds can be reached.

In this work we present a VP containing a novel ARMv8 SystemC Transaction Level Modeling 2.0 (TLM) compatible processor model. The model leverages the ARM Virtualization Extensions (VE) via the Linux Kernel-based Virtual Machine (KVM) to execute the target software natively on an ARMv8 host. To enable the integration of the processor model into a loosely-timed VP, we developed an accurate instruction counting mechanism using the ARM Performance Monitors Extension (PMU). The requirements for integrating the processor model into a VP and the integration process are detailed in this work.

Our evaluations show that speedups of up to 2.57x over state-of-the-art DBT-based simulator can be achieved using our processor model on ARMv8 hardware.

***Index Terms*—SystemC, TLM, ESL, KVM, Virtualization**

I. Introduction

With the technological advances predicted by Moore's law, Hardware/Software (HW/SW) systems have become more and more complex over the last decades. Today, it is common for systems to consist of many processors, accelerators and peripherals. Even small embedded systems run complex software stacks, often consisting of millions of lines of code. At an average error rate of one to two errors per hundred lines of code, this adds up to thousands of errors [1]. VPs, using TLM and its loosely-timed coding style [2], have become an essential tool for finding and fixing these errors in early phases of the design cycle. For example, they are used in the automotive industry to increase system test throughput by significantly reducing test setup and test analysis overhead in comparison with physical prototypes [3]. In addition, they can be used for HW/SW codesign, architecture exploration and design verification [4].

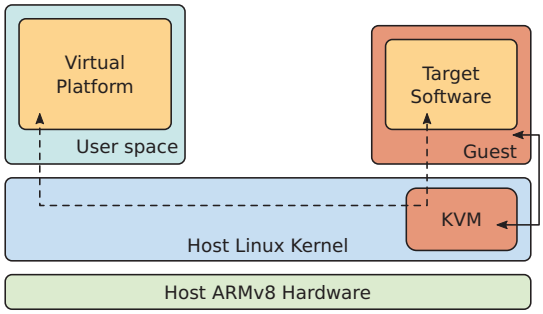

Fig. 1. ARM-on-ARM overview

For efficient employment of a VP in the design cycle, rapid simulation speed is paramount, e.g. when debugging a software problem that only arises after a certain runtime. When executing the target software, such as the Linux kernel, the performance of the Instruction Set Simulator (ISS) dominates the performance of the VP, since comparably little execution time is spent in other components. State-of-the-art solutions use DBT-based ISSs to emulate the target Instruction Set Architecture (ISA) on the host ISA, e.g. ARMv8 on x86-64. This incurs a significant performance overhead, since the target instructions are translated to the host ISA at runtime and a one-to-one instruction translation is rarely possible.

Virtualization enables encapsulated, native execution of software beyond userspace at near native speed [5]. It is available for the ARMv8 architecture using the ARM VE [6]. Here, ARMv8 code can be executed on the ARMv8 host processor in a special guest mode. This is enabled by an additional Exception Level (EL) [7]. When using the ARM VE the Linux kernel is executed in EL2 with the highest privileges, while guest code is executed in EL1 and EL0 with lower privileges. This way encapsulation is ensured. KVM offers a Linux kernel API to setup and execute code in guest mode on ARM hardware.

The main idea of the proposed approach is depicted in Figure 1. In this work we use the ARM VE via KVM to execute the target software of an ARMv8 VP natively on ARMv8 host hardware eliminating the need for DBT.

The contributions of this work are:

978-3-9819263-4-7/DATE20/©2020 EDAA

- A high-performance SystemC/TLM-compatible ARMv8 processor model using KVM for native execution
- An instruction counting mechanism for loosely-timed SystemC/TLM simulation with the processor model using the ARM PMU
- An ARMv8 VP using the novel processor model to evaluate representative benchmarks

II. Related Work

In the context of VPs mainly two types of ISSs are common. First, cycle-accurate simulators such as GEM5 [8] are used when high modeling detail is needed. In these simulators low-level behavior such as caching or branch prediction is reproduced. Due to the high modeling detail, cycle-accurate simulators do not offer sufficient performance for interactive simulation of complex target software. Hence, instruction-accurate simulators exist, which focus on modeling the behavioral aspects of the target ISA. These simulators offer sufficient modeling detail and simulation performance for software development and verification. Nowadays, many state-of-the-art instruction-accurate system simulators incorporate DBT-based ISSs [9]–[11]. These simulators are fast enough to execute complex target software, such as rich OS kernels, at reasonable speeds. In addition they offer deep introspection into the processor state which is advantageous for debugging. However, they still incur the performance overhead associated with DBT, which limits the achievable simulation speed.

Virtualization hardware extensions, such as ARM VE, have been used in the system simulation context to improve the simulation performance. For example, QEMU [9] offers a KVM backend as an alternative to DBT that enables native code execution via virtualization if KVM is available. It is important to note, that the use of KVM implies the same ISA for target and host. Since QEMU simulations are untimed, no timing information can be gathered from such a simulation. Also QEMU is CPU centric and does not offer a SystemC/TLM interface to connect other simulation building blocks, which limits interoperability.

Native simulation approaches [12]–[14] aim to improve simulation speed by eliminating the instruction translation costs at runtime as well as the induced overhead due to inefficient translations. In these works the target software is compiled to the host ISA and executed directly on the host. However, this requires the software to use specific APIs or a direct annotation of the software. Also it induces a decrease in simulation accuracy since target and host ISA differ. The mandatory availability of the target software source code is another restriction for this approach. The use of a combination of native execution and static binary translation to enable the simulation of VLIW processors has also been proposed [15]. By translating target binaries to native binaries the need for target source code can be eliminated, but the other limitations persist. Sandberg et al. proposed using KVM in combination with GEM5 to fast-forward target software execution to a region of interest for architectural simulation and performance estimation [16]. They note that the limited introspection into the processor caches when switching from KVM to GEM5 limits the accuracy of this technique. Nevertheless, a significant speedup was demonstrated.

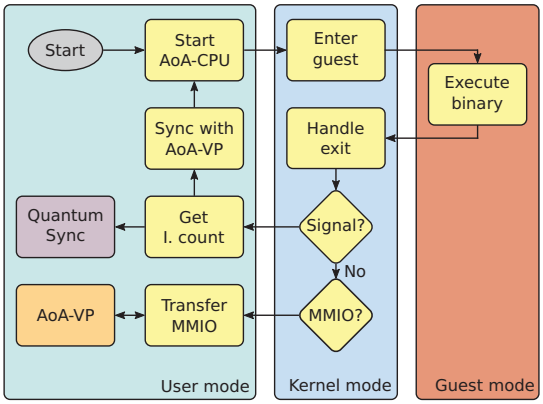

Fig. 2. Execution loop of *AoA-CPU*

In summary, it was shown that virtualization technologies have been used to improve the performance of system simulations. They do not offer the same level of introspection as DBT-based solutions, which can be disadvantageous in some scenarios.

III. AoA-CPU: An ARM-on-ARM Processor Model for Virtual Platforms

Our ARM-on-ARM (AoA) processor model, *AoA-CPU*, is a SystemC/TLM compatible ARMv8 processor model that executes the target software of an ARMv8 VP on ARMv8 hardware natively. Thus, the target software does not need to be recompiled for a different architecture and the binary does not have to be modified for execution on the VP. The VP is compiled for the ARMv8 architecture, then native execution is achieved by utilizing the ARM VE via KVM.

An overview of *AoA-CPU*'s execution loop is shown in Figure 2. It can be observed that the functionality of the model is distributed on all privilege levels from user to guest mode. Before the run loop begins, *AoA-CPU* instantiates a KVM virtual machine with a virtual CPU, which is later used to execute the target software natively. The system's main memory is also mapped at this point. KVM has specific requirements regarding its memory mapping which are described in Section III-A. The target software is executed in guest mode by KVM, which itself is part of the Linux kernel. To interact with KVM the corresponding API is used from user mode which is also where the remainder of the VP is executed.

Since the simulation is loosely-timed, the processor is allowed to run ahead of the global simulation time. This amount of time is referred to as the *quantum*. The processor executes instructions at a predefined clock speed, therefore the quantum can also be expressed in terms of instructions:

$$instructions\ per\ quantum = quantum \cdot clock\ rate$$

Since KVM does not provide a method for executing a predefined number of instructions, the *QuantumSync* instruction

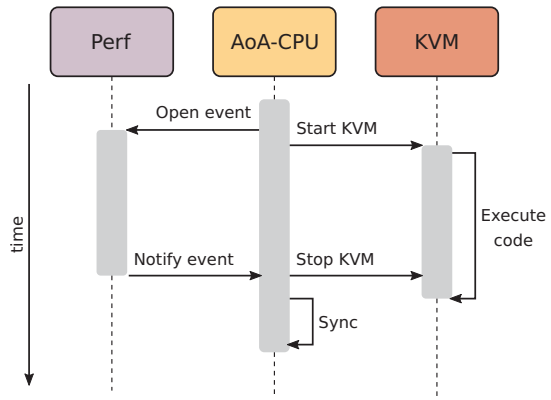

Fig. 3. Overview of instruction counting mechanism

counting mechanism based on Linux perf and the ARM PMU was implemented. *QuantumSync* is described in Section III-B.

During execution of the VP, the *AoA-CPU* run loop is triggered for every simulation quantum. KVM will execute as many target instructions as possible until it is stopped by *QuantumSync* or when interactions with other components of the VP occur. These interactions are typically interrupts or Memory-Mapped Input Output (MMIO) accesses. Interrupts have to be signaled from the VP to the processor model via KVM. The interrupt mechanism is introduced in Section III-D. MMIO accesses stop KVM and are then transfered to the rest of the VP, which is described in Section III-C.

In addition, a full GNU Debugger (gdb) integration is available for *AoA-CPU* which allows interactive debugging of target software.

A. Memory

AoA-CPU requires access to the VP's main memory for target software execution. For this the SystemC/TLM Direct Memory Interface (DMI) is used, which allows the processor model to map memories directly and access it via a pointer. This DMI pointer is made available to *AoA-CPU* in the setup phase of the VP and is then passed to KVM for memory mapping. DMI invalidations at runtime could be forwarded to KVM, as it allows for changing the memory mapping at runtime. However, this is left for future work. By setting up the memory as described above, it can now be accessed directly from the KVM virtual CPU without leaving the guest or interrupting target code execution.

B. The QuantumSync Instruction Counting Mechanism

A processor model in a loosely-timed SystemC simulation executes a predefined number of instructions for every quantum as explained at the beginning of this section. KVM does not offer to execute a fixed number of instructions, instead it will run until it is disrupted, e.g. by an MMIO access or an interrupt. At first glance this renders KVM unsuitable for integration into a loosely-timed SystemC simulation. However, KVM can be interrupted externally since it will stop execution at the reception of an unmasked Linux signal. In this work, the ARMv8 PMU in combination with Linux perf is used to generate this signal. The PMU is an optional hardware extension that equips the processor with a set of non-invasive debugging components that permit the observation of data and program flow inside the processor [7]. Among these components is an accurate cycle counter that can be used for counting instructions executed on the processor. The other component of the instruction counting mechanism is the Linux perf profiling tool. It can be used in conjunction with the PMU to count only the instructions that are executed in guest mode which are relevant for the quantum. To enable perf to count instructions executed in guest mode, a patch was applied to the Linux kernel[1].

An overview of the mechanism is depicted in Figure 3. First, *AoA-CPU* sets up a perf event for counting guest instructions that will send a signal when a target instruction count is reached. This number is equal to the number of instructions that should be executed in this quantum. When the signal is received by KVM, target software execution is stopped. Now the actual instruction count is retrieved from perf. This count can be slightly different than the target instruction count, since the perf signal requires some cycles to reach KVM. Finally, time is synchronized to the global SystemC time using the retrieved instruction count. The accuracy of this instruction counting method is evaluated in Section V-A.

C. Memory-Mapped Input Output

MMIO facilitates communication between *AoA-CPU* and the peripherals of the VP. When an MMIO access occurs, KVM will interrupt target software execution, as this is an unmapped memory access. *AoA-CPU* will handle this exit as depicted in Figure 2. It will gather the necessary information about the memory access, such as address, data, and whether a read or a write has occurred. From this information a TLM generic payload is constructed which is sent out via the TLM blocking transport interface. After satisfying the MMIO request, *AoA-CPU* will synchronize with the SystemC global simulation time by acquiring the current instruction count from *QuantumSync* and executing a SystemC wait. Then target software execution is resumed.

D. Interrupts

During VP simulation interrupts will occur which the processor model has to react to. In the ARMv8 architecture there are two main interrupts to the processor: the normal IRQ interrupt, and the fast FIQ interrupt. Usually these interrupts are connected to an interrupt controller such as the ARM Generic Interrupt Controller (GIC). Peripherals, e.g. timers or UARTs, signal their interrupts to the GIC which in turn raises the interrupt of the processor. For example, the ARM Generic Timer is used by the Linux kernel to generate a periodic interrupt for the scheduler. KVM allows to place the interrupt controller in user space or in kernel space. The first option fits well into a SystemC VP, because it allows to integrate the

[1]https://patchwork.kernel.org/cover/10874767

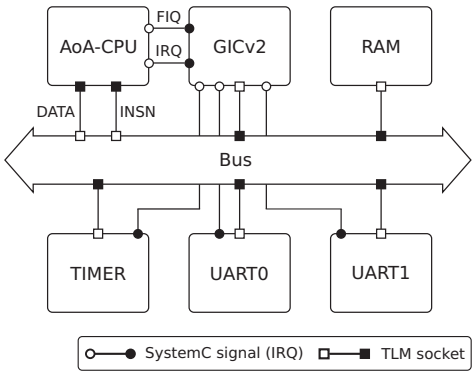

Fig. 4. AoA-VP overview

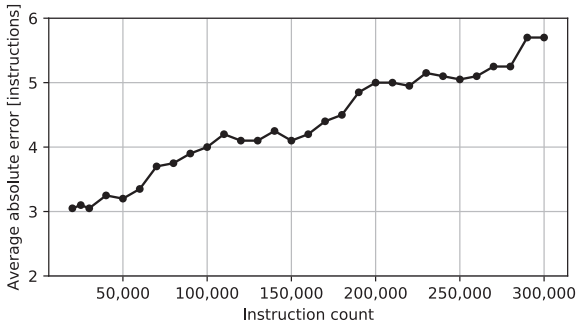

Fig. 5. Average absolute error of instruction counting mechanism

processor model with a TLM interrupt controller model. This offers more introspection than placing the interrupt controller in kernel space. Interrupts can be propagated from peripherals to the interrupt controller and then to the processor model via a SystemC signal. In *AoA-CPU* these SystemC signals are used with the KVM API to signal interrupts to the KVM virtual CPU which reacts to the interrupt signal accordingly.

IV. AoA-VP: A Fast ARM-on-ARM Virtual Platform

The ARM-on-ARM Virtual Platform (AoA-VP) was constructed to demonstrate the capabilities and evaluate the performance of the novel processor model. An overview of AoA-VP is shown in Figure 4. Besides the *AoA-CPU* it contains several other peripherals:

- An ARM GICv2 interrupt controller model
- A memory model
- A timer model
- Two instances of a UART model
- A bus model

The peripheral models are connected to a bus model via SystemC/TLM sockets. Peripheral interrupts are sent to the GICv2 via SystemC signals. The GIC then controls the FIQ and IRQ signals of the processor. AoA-VP is designed for executing the CoreMark benchmark and booting the Linux kernel. For the CoreMark benchmark the interrupt controller is not used and the timer model provides access to the wall-clock time. When Linux is executed, the timer model implements a memory-mapped ARMv8 Generic Timer, which is used by the Linux kernel for generation of the scheduler interrupt. The GICv2, memory, UART, and bus model are part of Virtual Components Modeling Library[2], while the timer was implemented as part of this work.

V. Experimental Evaluation

Several experiments were conducted to evaluate the performance of *AoA-CPU* in the AoA-VP. First, the accuracy of the instruction counting mechanism was analyzed, since this mechanism is crucial for exact simulation and performance evaluation. Next, the CoreMark benchmark and the Linux boot time were evaluated as performance indicators. CoreMark was selected to quantify *AoA-CPU's* performance since it is a standard CPU benchmark. It is small in size and the Memory Management Unit (MMU) is not used. DBT-based simulators typically perform well on this workload since the entire translated program can be cached and no MMU simulation overhead is incurred. In addition the Linux kernel boot time was added as a performance metric, since it is control flow driven and utilizes the MMU. In DBT-based simulators, simulating an MMU adds overhead to the processor model, because page table walks are executed for every uncached address translation. Unlike DBT-based simulators *AoA-CPU* utilizes the host MMU. Therefore, no MMU simulation is required.

For comparison the identical target binaries were executed on Synopsys Virtualizer O-2018.09-SP1 [10] using the ARM Fast Models. Since *AoA-CPU* is executed on ARMv8 hardware and Virtualizer on x86-64, CoreMark was also executed natively on both platforms to ascertain what percentage of the native performance can be utilized in the VP. A Socionext SynQuacer Developerbox containing a Socionext SC2A11 SoC with 24 ARM Cortex-A53 cores clocked at 1 GHz running Linux 5.1.0-rc2 was used to execute AoA-VP. Virtualizer was executed on an Intel Xeon E5-2687W v4 server machine clocked at 3 GHz running CentOS 6.6. The results of these evaluations are presented in the following sections.

A. Accuracy Instruction Counting

To evaluate the accuracy of the instruction counting mechanism, a program containing a loop was executed using AoA-VP. The number of executed loop iterations is counted in a register:

```
_loop :
  add x2, x2, #1
  cmp x2, x1
  ble _loop
```

Target software execution is then interrupted by the signal generated by perf as explained in Section III-B. Since the

[2]https://github.com/janweinstock/vcml

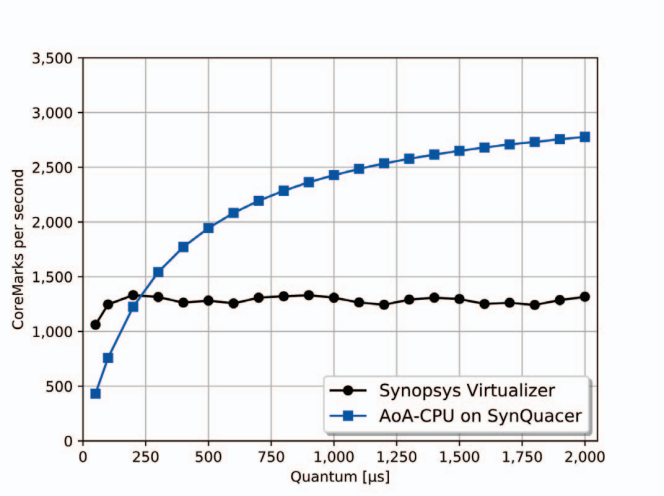

Fig. 6. CoreMark performance comparison

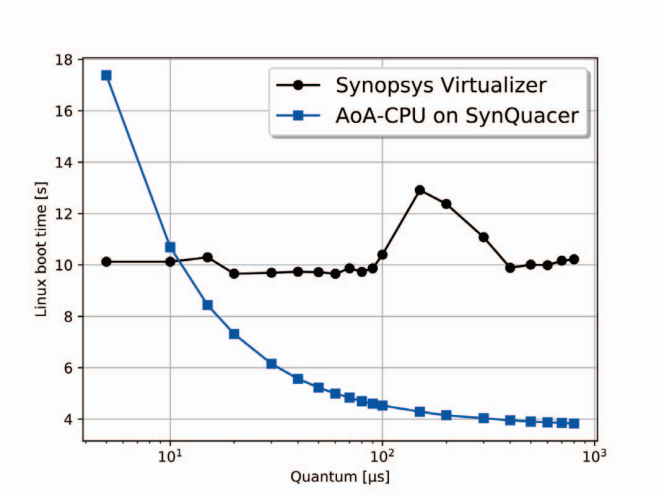

Fig. 7. Linux boot time comparison

number of instructions in one loop iteration is known, the number of actually executed instructions can be calculated as:

$$count = loops \cdot \frac{instructions}{loop} + prior\ instructions$$

Because the loop can be interrupted at any instruction an error of two instructions cannot be eliminated. Next, the instruction count measured with perf is retrieved. The difference between these two instruction counts constitutes the error in the instruction counting mechanism. Figure 5 shows the average absolute error in the perf instruction count for different target instruction counts. Even though there is a slight increase in the absolute error with rising target instruction counts, the overall error in the instruction counting is very low ($<0.01\%$). A slight quantum overshoot occurs, which is generally unproblematic.

B. CoreMark Performance

The CoreMark benchmark was executed on AoA-VP and Synopsys Virtualizer for different SystemC/TLM quanta. Figure 6 presents the results of this evaluation. It can be observed that for quanta larger than $300\,\mu s$ *AoA-CPU* outperforms Synopsys Virtualizer. For lower quanta the overhead of context switching between *AoA-CPU* in guest mode and the remaining VP dominates the performance. The peak performance of *AoA-CPU* is 2778 CoreMarks/s which corresponds to 506 Million simulated Instructions Per wall-clock Second (MIPS). *AoA-CPU* MIPS can be calculated accurately, since the exact number of executed instructions for the target binary is known from the Virtualizer simulation.

In total, an average speedup of 2.08x over Virtualizer was achieved, even though Virtualizer is executed on significantly more powerful hardware. *AoA-CPU* performs better than Virtualizer, because the benchmark is executed natively on the processor. Virtualizer on the other hand incurs the DBT overhead of the ARM Fast Models as explained in Section I.

C. Linux Boot Time

Linux boot time was selected as a benchmark for a realistic software workload for AoA-VP. The time from simulation start to login shell was measured for different SystemC/TLM quanta. Figure 7 summarizes the results of this evaluation. Linux boots faster on *AoA-CPU* than on Synopsys Virtualizer for quanta larger than $15\,\mu s$. A peak performance of $3.72\,s$ boot time was achieved which corresponds to a speedup of 2.57x over Virtualizer. This speedup was achieved even though *AoA-CPU* is executed on less powerful hardware. For smaller quanta the performance is again dominated by the expensive context switches between *AoA-CPU* in guest mode and the remaining VP.

It can be observed, that for the Linux boot *AoA-CPU* provides a higher speedup over Virtualizer than for CoreMark. In Virtualizer the MMU of the processor has to be simulated, whereas *AoA-CPU* uses the physical MMU. Hence, address translations with page table walks as they are used by Linux are more expensive on Virtualizer because they are not accelerated by hardware.

There is a performance decrease for quanta larger than $100\,\mu s$ on Synopsys Virtualizer using the ARM Fast Models. Since both components are proprietary the reason for this slowdown could not be further investigated.

D. Native Performance Ratio

Since the Socionext SynQuacer hardware executing AoA-VP is different from the hardware executing Synopsys Virtualizer, an evaluation of the native CoreMark performance was conducted. For this, CoreMark was compiled for the x86-64 architecture of the Xeon host of Synopsys Virtualizer and for the ARMv8 Socionext SynQuacer. Then the CoreMark benchmark was executed natively on each platform. The results of this evaluation are summarized in Table I. On average, on the Intel Xeon hardware, a score of 16 159 CoreMarks/s was achieved. The Socionext SynQuacer achieved a score of 3294

TABLE I
COREMARK PEAK PERFORMANCES (PP) COMPARISON.

Technology / Platform	Simulation PP	Native PP	Ratio
Synopsys Virtualizer / Intel Xeon	1332	16 159	8%
AoA-CPU / SynQuacer	2778	3294	84%

CoreMarks/s. This is due to the 3x higher clock frequency and completely different architecture of the Intel Xeon. The result indicates, that the Intel Xeon processor is more powerful and capable of executing more instructions in a given time than the Socionext SynQuacer.

In the Synopsys Virtualizer VP running on the Intel Xeon, a performance of 1332 CoreMarks/s was reached. Here the ARMv8 target software instructions are emulated using the ARM Fast Models. *AoA-CPU* achieved a score of 2778 CoreMarks/s on the Socionext SynQuacer whilst executing the same target binary.

Due to the overhead of the DBT in the ARM Fast Models, the Synopsys Virtualizer VP is only capable of reaching 8% of the native performance. *AoA-CPU* on the other hand is able of reaching 84% of native performance leveraging the ARM Virtualization Extensions on the Socionext SynQuacer. This makes our proposed methodology 10x more efficient.

However, it has to be noted that *AoA-CPU* cannot execute a hypervisor or a trusted execution environment. Also it is not as extensible as a DBT-based simulator, since the target software is executed natively. For example, it is currently not possible to extend the instruction set of *AoA-CPU*.

VI. CONCLUSION

This paper revealed that utilizing virtualization hardware extensions, such as the ARM VE, in a SystemC/TLM simulation is viable, fast and accurate. For this, a prototype implementation was presented including a SystemC/TLM compatible processor model, *AoA-CPU*, that utilizes said hardware extensions via KVM on modern ARMv8 hardware. With *QuantumSync*, a solution to the problem of instruction counting inside the KVM guest was proposed and its accuracy evaluated. In addition, AoA-VP, a VP containing the *AoA-CPU*, was developed for verification and performance evaluation. Since ARMv8 hardware extensions are used, the processor model and VP are compiled for the ARMv8 architecture and executed on ARMv8 hardware.

Two representative workloads were selected as performance indicators: the CoreMark CPU benchmark and the Linux kernel boot. AoA-VP with *AoA-CPU* outperformed the state-of-the-art Synopsys Virtualizer system simulator containing the ARM Fast Models by a factor of up to 2.08x and 2.57x respectively for the aforementioned workloads. This speedup was achieved even though Virtualizer was executed on more powerful hardware.

The ARMv8 hardware utilized for benchmarking AoA-VP is available for less than half of the price of the hardware utilized for the Virtualizer benchmarks, making our proposed methodology not only faster but also more cost efficient than state-of-the-art DBT-based simulators.

Since AoA-VP is targeted for software development, a full kernel mode debug integration is available in *AoA-CPU*. When debugging, there is no discernible difference between AoA-VP and physical hardware.

ACKNOWLEDGEMENTS

This work has been partially supported by AUDI AG.

REFERENCES

[1] J. K. Horner and J. Symons, "Understanding Error Rates in Software Engineering: Conceptual, Empirical, and Experimental Approaches," *Philosophy & Technology*, Feb 2019, springer Netherlands. [Online]. Available: https://doi.org/10.1007/s13347-019-00342-1

[2] IEEE, "IEEE Standard for Standard SystemC Language Reference Manual," *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, Jan 2012.

[3] S. Bayon de Noyer and X. Pan, "ADAS SW development and integration on a Virtual Prototype at NXP and Tier 1," in *Proceedings of the 2018 Synopsys User Group (SNUG) China*, 2018.

[4] S. Vinco and N. Bombieri and D. Pagliari and F. Fummi and E. Macii and M. Poncino, "A Cross-level Verification Methodology for Digital IPs Augmented with Embedded Timing Monitors," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 24, no. 3, 2019.

[5] J. Che and C. Shi and Y. Yu and W. Lin, "A Synthetical Performance Evaluation of OpenVZ, Xen and KVM," in *2010 IEEE Asia-Pacific Services Computing Conference*, Dec 2010, pp. 587–594.

[6] Dall, Christoffer and Nieh, Jason, "KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 333–348.

[7] ARM Limited, "ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile," Issue E.a.

[8] C. Menard and J. Castrillon and M. Jung and N. Wehn, "System simulation with gem5 and SystemC: The keystone for full interoperability," in *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, July 2017.

[9] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator." in *USENIX Annual Technical Conference, FREENIX Track*, vol. 41, 2005.

[10] "Synopsys Virtualizer," https://www.synopsys.com/verification/virtual-prototyping/virtualizer.html, accessed: 2019-03-22.

[11] "Open Virtual Platforms," http://www.ovpworld.org/, accessed: 2019-03-23.

[12] H. Shen, M. Hamayun, and F. Petrot, "Native Simulation of MPSoC Using Hardware-Assisted Virtualization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 7, pp. 1074–1087, July 2012.

[13] Jerraya, Ahmed A. and Wolf, Wayne, "Hardware/Software Interface Codesign for Embedded Systems," *Computer*, vol. 38, no. 2, pp. 63–69, Feb. 2005.

[14] D. Mueller-Gritschneder and A. Gerstlauer, *Host-Compiled Simulation*. Springer Netherlands, 2017, pp. 593–619.

[15] M. Hamayun, F. Pétrot, and N. Fournel, "Native simulation of complex VLIW instruction sets using static binary translation and Hardware-Assisted Virtualization," in *18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2013, pp. 576–581.

[16] A. Sandberg, N. Nikoleris, T. E. Carlson, E. Hagersten, S. Kaxiras, and D. Black-Schaffer, "Full Speed Ahead: Detailed Architectural Simulation at Near-Native Speed," in *IEEE International Symposium on Workload Characterization*, Oct 2015, pp. 183–192.