Impact of Performance Estimation on Fast Processor Simulators

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

Friedrich-Alexander-Universität Erlangen-Nürnberg,
Martensstr. 3, 91058 Erlangen, Germany
{sebastian.rachuj,dietmar.fey,marc.reichenbach}@fau.de

Abstract. Hardware simulation is always a compromise between the speed of the simulator and the accuracy of the estimated runtime on the real hardware. Instrumenting fast simulation frameworks to estimate runtimes always results in tremendous slow-downs. In this paper, a quantization is done regarding the minimal overhead that can be expected when adding architectural models to a fast JIT enhanced emulation. Previous work is only focused on new approaches and improving available methods, but not on the unavoidable overhead that is introduced with any kind of instrumentation. Thus, the additional simulation time of calling an empty stub function instead of a fully implemented architectural model is investigated. We show relative runtimes for calling a function after executing each instruction and after executing a block of instructions. Also, a comparison against fully implemented models is done. On the test platforms, an academic and a commercial processor simulator were evaluated. The resulting average relative runtimes of the minimal introduced overhead are determined to be between 2.24 and 8.09 meaning that an emulation takes twice to eight times as long with instrumentation enabled.

1 Introduction

Processor Simulators are the means of choice when it comes to designing and evaluating new hardware systems. They not only provide the possibility to run the software before the silicon is available [3]. Additionally, design space explorations to find the best processor configuration for a specific application can be performed [15]. However, they always have to make a compromise between maximum simulation speed and accuracy [10]. Therefore, many different approaches were developed and investigated trying to optimize one dimension without sacrificing the other one. A brief overview about them is presented in Sect. 2.

The most promising method for creating a fast and precise simulator is by starting with a fast instruction set simulator (ISS) and adding additional models to make it more and more accurate [22]. Just-In-Time (JIT) compiling emulators are the fastest known ISS due to the fact that they translate the target machine code (meaning the machine code that runs on the ISS) to the host machine code. This reduces the overhead tremendously since the generated host code can be

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2021
Published by Springer Nature Switzerland AG 2021. All Rights Reserved
H. Song and D. Jiang (Eds.): SIMUtools 2020, LNICST 370, pp. 79–93, 2021.
https://doi.org/10.1007/978-3-030-72795-6_7

run directly on the host processor of the emulator as soon as the translation step finishes. Prominent examples for JIT compiling emulators include the Tiny Code Generator (TCG) backend of QEMU [4], the code morphing facility of OVPsim [12], and the ARM Fast Models [2].

While these emulators are capable of executing software with very high speed, they lack means of determining nonfunctional properties like performance and energy requirements. There are plugins and extensions enabling these simulators to perform estimations of the runtime behavior. However, each activated model reduces the runtime since additional aspects of the hardware (e.g. the pipeline of the processor) have to be simulated. Even without modelling minor details, the runtime is impacted just by having to run further code next to the translated host code.

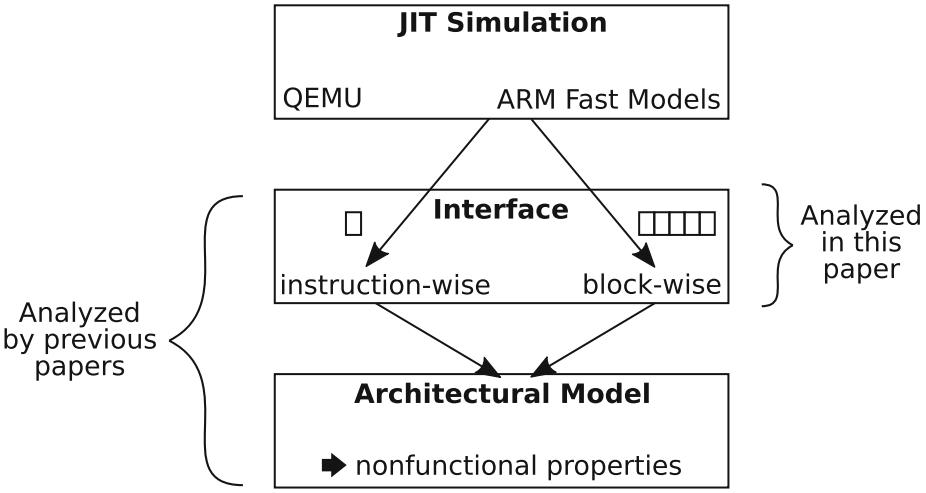

Fig. 1. The usual layout of a JIT emulator that is enhanced by architectural models. The analyzed part of this paper and parts that were previously evaluated are marked with curly braces.

The purpose of this paper is to quantize the minimal overhead that can be expected when adding functionality for estimating nonfunctional properties to a JIT compiling emulator. Figure 1 shows the usual parts of a fast simulation that is enhanced by an architectural model to estimate nonfunctional properties like time and energy. From the JIT accelerated emulation, an interface is used that transfers the information about executed instructions to the model. This is often done using callback mechanisms that are used to run a function for each instruction. While the architectural model can be adjusted with a high degree on variation depending on the requirements, the interface forwarding the information from the JIT simulation to the architectural model allows only minor modifications. Thus, the interface limits the maximum reachable instruction throughput and is therefore evaluated in this paper. Prior work only focused

on a complete architectural model including the interface and did not analyze the interface separately. For this paper, QEMU as an example for an academic simulator and the ARM Fast Models as an example for a commercial simulator are investigated. While the ARM Fast Models already provide a probing infrastructure, QEMU had to be extended to support analyzing the instruction stream during the emulation process. For a faster execution, it is possible to forward multiple instructions at once in the form of translation blocks or basic blocks. This is an optimization to reduce the investigated overhead while still providing all data that is required to drive an accurate model. In Fig. 1 the two methods are called instruction-wise for running the callback for each emulated instruction and block-wise for running the callback for multiple instructions. In order to classify the additional overhead, comparisons to fully modelled architectural simulations are presented later in this paper. For QEMU, this consists of a literature review since there is no architectural model directly available for QEMU. On the other hand, the ARM Fast Models provide a pipeline implementation that was deployed and its overhead quantified.

This paper is structured as follows. At first, a brief overview is given about different methods for fast and accurate simulation. Afterwards, the simulation infrastructure for measuring the overhead is presented in Sect. 3. The findings of this paper are shown in Sect. 4. Finally, a conclusion is drawn.

2 Related Work

Many different methodologies for implementing precise and fast simulators were published. However, none of these publications investigated the minimal required overhead when fast JIT compilers have to be extended to support mechanistic or statistical models. In this section, some of these approaches are presented.

Multiple papers were written about extending QEMU with the ability to measure non-functional properties. QSim is a very promising approach combining QEMU with sim-outorder which is part of SimpleScalar [6,16]. It achieves huge speeds while still modelling the architectural aspects cycle-accurately by caching the results of the sim-outorder model of SimpleScalar for each translation block. However, small deviations in accuracy can be observed since effects spanning multiple translation blocks are not correctly modelled.

TQSIM is another way for extending QEMU with runtime estimation [13]. Here, a sampled approach has been implemented, based on an analytical model and sampled simulation, allowing a high speed simulation in comparison to a cycle-accurate simulator. On the other hand, they experience a slowdown from 3 to 14 in comparison to plain QEMU. Their accuracy is still very high with an error of only 8%.

A plugin interface for the code translator of QEMU (TCG) was recently added to the emulator[1]. It offers isolated functions for manipulating the intermediate code generation. Like implemented in this paper, it is also possible to add

[1] In git commit 68d8ef4ec540682c3538d4963e836e43a211dd17.

callback functions for each instruction or translation block and even to generate certain immediate instructions like an add on a custom virtual register. This can be used to implement an instruction counter. For this paper, the plugin system was not used as it is not as flexible as direct code changes, yet. Additionally, for interfacing with SystemC, code changes were necessary nonetheless.

In addition to the already mentioned simulator extensions that make functional simulation more accurate like statistical simulation and integration of architectural simulators, there are many more approaches for speeding up simulators while retaining a high accuracy. This can be done by parallelizing the simulation as done for example with PQEMU [8]. If the host and guest instruction set architecture is the same, even near-native speed for simulation can be achieved by exploiting parallelism and the virtualization capabilities of a platform [20]. Additionally, new simulators are also created that try to achieve new speed records when modelling a pipeline [9]. However, none of these publications contain an analysis of the minimal expected overhead with an empty model.

The peripheral system used for evaluation within this paper is implemented in SystemC which also required adjustments in QEMU. This has been done multiple times before. QBox is an example that uses a custom C-style TLM-2.0 library which is similar but not completely compatible with existing C++-style TLM-2.0 compatible models [7]. A transactor module is required to translate C-style transactions into C++-style transactions. Further implementations that are compatible to TLM-2.0 as defined in the SystemC standard include an implementation called LibSystemCTLM-SoC of Xilinx and QEMU-SystemC [18,23]. Xilinx's library uses a remote socket protocol to connect SystemC and QEMU while QEMU-SystemC is integrated into QEMU. These libraries are oftentimes more difficult to use and don't get the newest patches of QEMU. This means that new features like the RISC-V frontend is missing when using them.

Publications featuring the ARM Fast Models are sparsely found. Nonetheless, there are implementations of architectural models that enhance the Fast Models by timing behavior estimation. Available aspects that can be modelled include cache simulators and a pipeline behavior simulator. However, an analysis of the overhead introduced by the different available probes and models is not yet publicly available.

Other emulation frameworks were also extended with nearly cycle-accurate models. Examples include the works of Rosa et al. and Schreiner et al. who implemented a more accurate simulation on top of OVPsim [19,21]. But, they also did not evaluate the overhead in comparison to an OVPsim without any instrumentations.

JIT compilation techniques are also used in dynamically executed languages like Java, Python and JavaScript to implement their virtual machines. In this area, the most important features to add to the code generation are control flow analysis and execution tracking. This is done for debugging purposes or to offer additional security functionality. Kerschbaumer et al. adopted the JavaScript engine of WebKit to track the data flow within the programs and to check, if sensitive data is sent to an untrusted domain [14]. While they measured an

overhead of 74%, the value is not comparable to emulation since direct changes of the JIT compilation infrastructure was done and additional control flow analysis integrated.

3 Simulation Infrastructure

For the investigation of this paper, two simulators capable of emulating the AArch64 instruction set architecture (ISA) have been investigated. Interconnecting peripherals in virtual hardware systems is commonly based on SystemC [5,11]. The abstraction level of virtual prototypes makes the TLM libraries of SystemC a sensible choice. However, the first considered simulator called QEMU, an academic and open source emulator [4], does not support interfacing with SystemC by default. Additionally, QEMU has no features for analyzing the instruction stream in more detail. The ARM Fast Models are a commercial simulator which is used within Platform Architect of Synopsys [3] for simpler usage. It supports SystemC by default and provides a probing infrastructure which is explained in the appropriate section. In the following, the adjustments that were made to the simulators to evaluate the introduced overhead are presented.

3.1 QEMU

QEMU is intended to be a fast emulation framework for running simple programs or even full operating systems. The high emulation speed is accomplished by using JIT compilation techniques meaning a translation of the guest machine code into the host machine code. While it also supports interpretation of its intermediate language and virtualization, these additional emulation modes were not of interest for this paper. Interpretation is slower than translation since more code has to be executed for each instruction. Virtualization requires the host to have the same ISA as the guest and is thus not possible if both ISAs do not match. QEMU's high speed makes it interesting for the evaluation. In the following, the execution system of the QEMU JIT backend is explained and the necessary changes to support callback functions that can be used to drive an architectural model are described.

Translation to the host machine code happens in two steps. First, the guest machine code is translated into an intermediate representation of the TCG component of QEMU. Afterwards, the resulting code can be optimized and translated again into the host machine code. A whole translation block containing multiple instructions is processed at once. It is a similar concept like basic blocks known for control flow analysis as required in optimizing compilers [1]. A basic block identifies a sequence of instructions that have only jumps to the beginning of the block and no jumps within the block with the exception of the last instruction. Translation blocks add additional constraints like a maximum length and a maximum of two successors. If a guest instruction has to be executed but the corresponding translation block is not yet within the code cache, a jump back

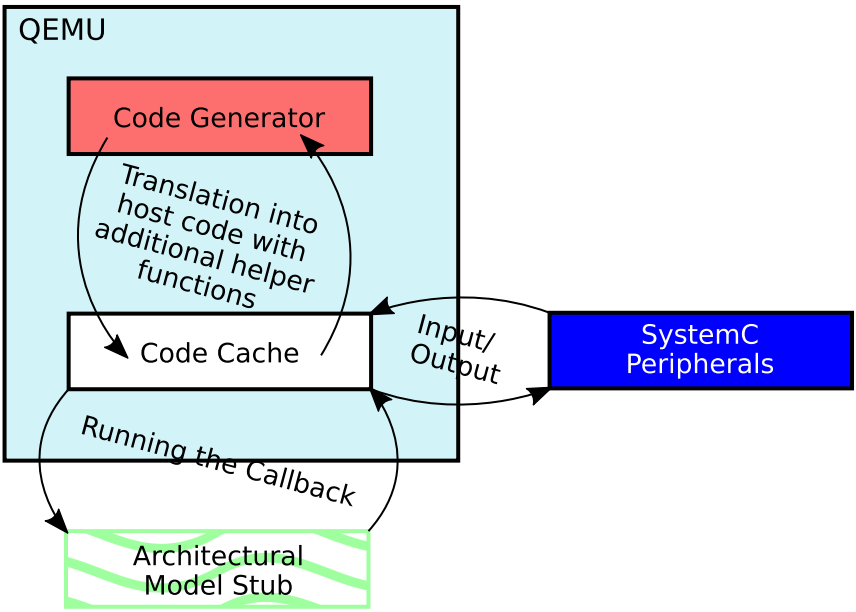

Fig. 2. Layout of the simulation framework used for this paper with QEMU as the binary translator. The code generator had to be adjusted and the peripherals and the architectural model stub were added for the evaluation.

into the code generator has to be done to translate the new translation block. This can be seen in Fig. 2.

Each guest instruction is translated into an arbitrary amount of TCG instructions. To ease implementing complex instructions, so-called helper functions can be defined. These functions can be implemented in high-level C and are called when the original guest instruction has to be emulated. While this functionality is normally used for implementing algorithmically complex guest instructions like cache maintenance or exception related operations, it is used to forward information about the execution flow to the architectural model. For this, the code generator had to be adjusted that a helper function is called for each executed guest instruction. Within this additional helper function, a detailed model can be driven. Since the aim of this paper is to evaluate the raw overhead of calling architectural functions, the model is just a stub as shown at the bottom of Fig. 2.

Since the overall simulated platform should stay the same for QEMU und the ARM Fast Models to allow an easier comparison, QEMU had to be extended with support for SystemC as also depicted in Fig. 2. For this, a custom machine type had to be implemented that is capable of interfacing with TLM2-compatible SystemC modules. It connects to plain memory by using the direct memory interface that offers a pointer referencing directly into the underlying memory. More sophisticated devices like the UART were implemented by mapping the hardware

registers with the help of so-called IO ranges. When a QEMU memory access goes into an IO range, a previously defined callback is executed. For this paper, we forward the information about the memory access to the SystemC modules using the blocking transport interface of TLM2. As already mentioned in Sect. 2, there are a lot of different approaches for connecting QEMU to SystemC. Since some of these implementations change QEMU tremendously and do not profit from changes of the original code base, the previously presented connection was introduced.

3.2 ARM Fast Models

The ARM Fast Models are intended to provide a fast processor simulator for hardware system generation that uses a JIT compilation technique [2]. It does not allow radical changes like the previously introduced changes to QEMU due to the source code not being publicly available. However, a set of interfaces is provided allowing the interaction with the models. The most important interface is the master socket that allows plugging in TLM2-compatible SystemC modules by using a bridge that is supplied with the Synopsys Platform Architect which is used as a development environment. In this way, the peripherals of the simulation framework can be used by the models as depicted in the lower part of Fig. 3.

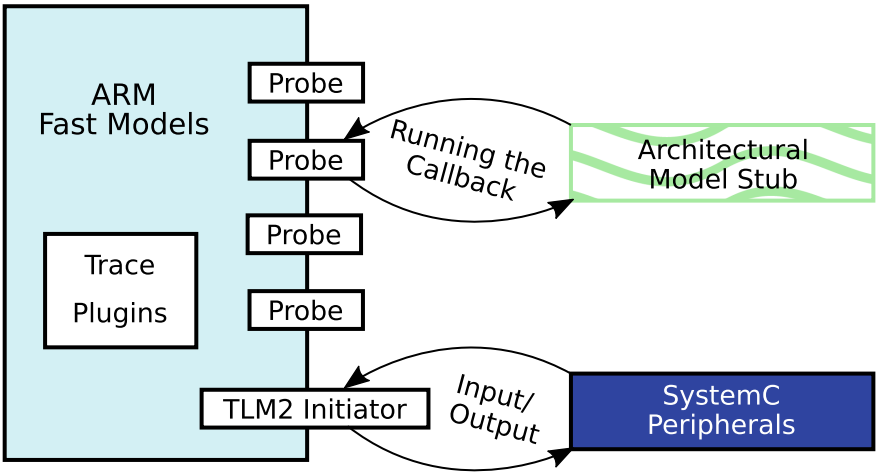

Fig. 3. Layout of the simulation framework used for this paper with the ARM Fast Models as the binary translator. The TLM2 initiator is offered by an additional bridge connected to the models. Again, the peripherals and the architectural model stub were added for the evaluation of this paper.

Additionally to the SystemC interface, the ARM Fast Models also support probes. They allow the registration of callbacks that are run on certain events during the simulation. For this paper, the `EndInstructionExecutionProbe` and

`BeginBasicBlockProbe` are of importance. The first one executes a callback every time the execution of an instruction finishes, while the second calls a callback as soon as a new basic block is entered. The interfaces use the observer pattern and the callbacks have to be implemented in C++. This allows driving the same architectural model stub as for QEMU. The overall layout of this simulation system is shown in Fig. 3. Usually, the implemented probes do not interact with the simulation except for gathering the information available via callbacks. This means that the information about nonfunctional properties is printed separately and is not returned to the emulation.

Another way to track the instructions is the usage of a trace plugin which can also be seen in Fig. 3. They are intended for creating custom tracing exports but can also be used for architectural models that are not implemented directly inside the ARM Fast Models. ARM delivers different plugins like a Tarmac Trace plugin and an example pipeline model. For this paper, the pipeline model was used to compare the measured callback overhead against the overhead of a full architectural simulation. All of the benchmarks were run with backdoor mode enabled meaning that the simulator uses the direct memory interface to access the memory.

3.3 Benchmarks

The benchmark programs that were used to measure the overheads introduced by architectural callbacks are the single-threaded RISC-V benchmarks[2]. They offer different kinds of algorithms and thus provide a variety of loads for the simulators. Since they have some simulator and architecture specific code, minor adjustments had to be made to make them compatible to the presented simulation infrastructure. Additionally, the runtime was increased by generating greater data sets. Table 1 shows the new data set size (depending on the algorithm, mostly meaning the size of the processed arrays) for each benchmark and gives a brief explanation about the algorithm that is implemented in the benchmark. Measuring the time of a benchmark run was done by the `setStats` function that is provided by the available code. For this, the `gettimeofday` function was modified to return a high precision wall-clock timestamp. All runs were bare-metal runs meaning that there was no operating system available in the simulated system. Only the plain execution of the algorithms was analyzed.

4 Results

Different overheads were analyzed for this paper. The baseline for comparison was measured by not having any callback executed during the simulation. This is the native speed of these simulators if just the emulation of the software is required. The first overhead that was measured is the one introduced by running a callback for all emulated instructions. As previously explained, the callback is

[2] https://github.com/riscv/riscv-tests/tree/master/benchmarks.

Table 1. Data size of benchmark programs

Benchmark	Description	Data Set Size
median	1D median filter	1048576
multiply	Inner product of two vectors	1048576
qsort	Quicksort (with insertion sort for small sizes)	1048576
rsort	Quicksort	1048576
spmv	Sparse matrix vector multiplication	R = 1024, C = 1024, NNZ = 65631
towers	Towers of Hanoi problem	20 discs
vvadd	Vector vector addition	1048576

only an architectural stub and does no actual modelling. This allows identifying the overhead of a simulator with additional instrumentation without also measuring the new functionality.

The second overhead that was investigated is based on callbacks that are only run for each translation block in case of QEMU or each basic block in case of the ARM Fast Models. This is a sensible optimization of the single-instruction callbacks since the instructions executed in all of these blocks are the same each time a certain block is emulated. A potential architectural model might profit from already having seen such a block before and might reuse older results from a previous call [16].

In the case of the ARM Fast Models, a third experiment for overhead measurement was performed. It includes a fully implemented pipeline model for a processor simulator. QEMU does not offer easily exchangeable architectural models but some related works are available as mentioned in Sect. 2. The findings of this paper are discussed and compared to the findings of these works.

All results presented in this section represent the average of at least 100 runs of the benchmark within the simulation. Before starting the simulation, the file system cache of the host operating system was warmed up with a few runs of the simulation. During the measurement no additional workload except from background processes was applied to the host system. The simulations took place on a standard desktop machine offering an Intel Core i5-7500 CPU. Results are given in the form of a relative runtime r that can be defined as seen in (1) with t_{base} being the measured time of the base simulation against which the comparison is made and t_{instr} being the measured time with additional instrumentation or pipeline models. This means that a relative runtime of one denotes no measurable overhead.

$$r = \frac{t_{instr}}{t_{base}} \tag{1}$$

4.1 QEMU

Figure 4 shows the results of the investigation in form of relative runtimes when QEMU runs the implemented callbacks. For each benchmark, two overheads, one for running a callback for each instruction and one running a callback for

each basic block, are given. The relative runtime ranges between 5.88 and 1.19 depending on the callback frequency and the benchmark. On average, the relative runtime amounts to 2.76 for instruction-wise processing and 2.24 for block-wise processing.

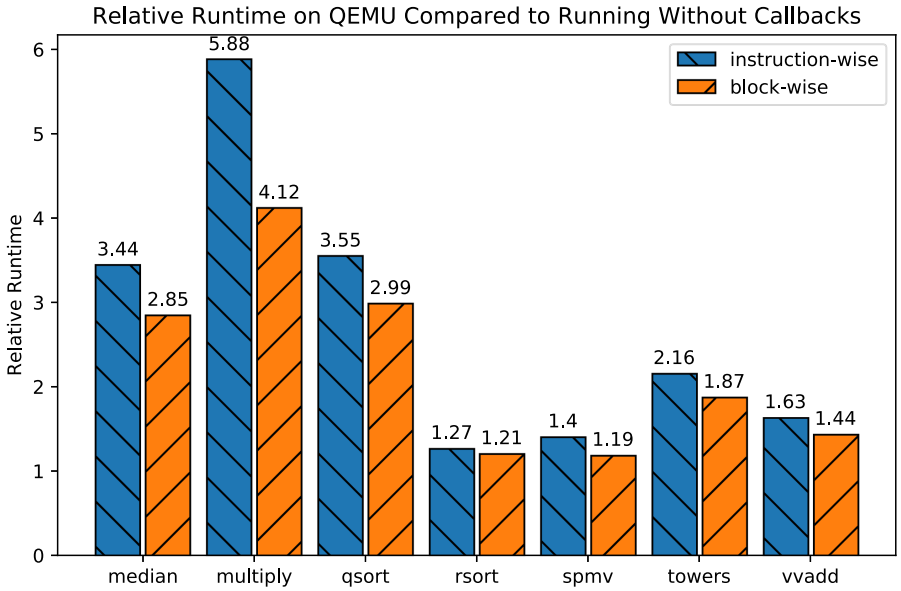

Fig. 4. The measured relative runtime for the different benchmarks when running on QEMU. Instruction-wise results were created by executing a callback for each emulated instruction, while block-wise results were created by executing a callback for each translation block.

The fluctuation between the different overheads of the benchmarks arises from the different instruction mixes of the programs. Memory-intensive applications that contain many load and store operations oftentimes have to call helper functions for performing the memory access task. Since this is already much overhead compared to arithmetic-intensive applications (like e.g. multiply), the additional overhead introduced by the callbacks to drive the architectural model is negligible. This can be seen for example by rsort and spmv which are very memory-intensive and thus have the lowest overhead. To prove this effect, two artificial benchmarks were created while one contains many more memory accesses than the other one. They showed exactly the previously described behavior.

Additionally, it can be seen in Fig. 4 that only executing one callback for each translation block reduces the overhead compared to running it for each instruction. This is due to the fact that less function calls have to be performed. Applications that have a higher overhead by instruction-wise instrumentation also benefit more from block-wise executed callbacks. If the instruction-wise callbacks are not very significant, running less of them can't result in a huge improvement. However, if the instruction-wise overhead is big, it is very noticeable when changing to block-wise callbacks.

Since there are no other architectural models available that can be directly measured, we will compare the findings against values found in publications. While for QSim no comparison against QEMU is given (only against sim-outorder), it is not suitable for giving an overhead as comparison [16]. TQSim on the other hand offers such a comparison from which we can conclude a relative runtime of 6.17 on average over all benchmarks [13]. However, since sampled instrumentation and statistical extrapolation is used, the overhead is expected to be less than having a callback executed for each instruction as evaluated in this paper. A more conventional approach is done by Miettinen et al. who extended QEMU with instruction classification, a cache and a TLB model [17]. When enabling all architectural aspects they implemented, the relative runtime in comparison to the simulation without instrumentation ranges from 27.25 to 29.33. With the findings of our paper, it can be said, that even with the fastest models, the average relative runtime will not be below 2.76 on average due to the added functions. Thus, the callback overhead is approximately 10% of the overall overhead introduced by the models of Miettinen et al. This means that one-tenth of the overhead originates from just transferring information into the architectural models. When optimizing the models, the interface will become responsible for a greater amount of overhead.

In QEMU, further optimizations to the overhead are possible. Instead of calling a helper function, the whole architectural model could be implemented using TCG intermediate code. This allows driving the model without any additional callback by integrating the code directly into the code cache. Additionally, it is possible that the created TCG code can be optimized by the optimization functionality of QEMU. However, this approach is very work intensive as it requires the model implementation to be written completely with a code that is similar to assembler languages. Additionally, it is most likely that an architectural model itself will call many further functions and thus cannot benefit much from this approach.

4.2 ARM Fast Models

The evaluation of the Fast Models showed a different behavior which can be seen in Fig. 5. Results are given as a relative runtime of executing a callback each instruction (instruction-wise) and executing a callback each basic block (block-wise) compared to running the simulation without any callback. With the exception of the SPMV benchmark, the overheads do not show significant differences ranging between 8.25 and 9.63. Additionally, the benefit from using

block-wise instrumentation is nearly the same for all investigated algorithms. On average, the relative runtime amounts to 8.09 for instruction-wise and 4.45 for block-wise instrumentation.

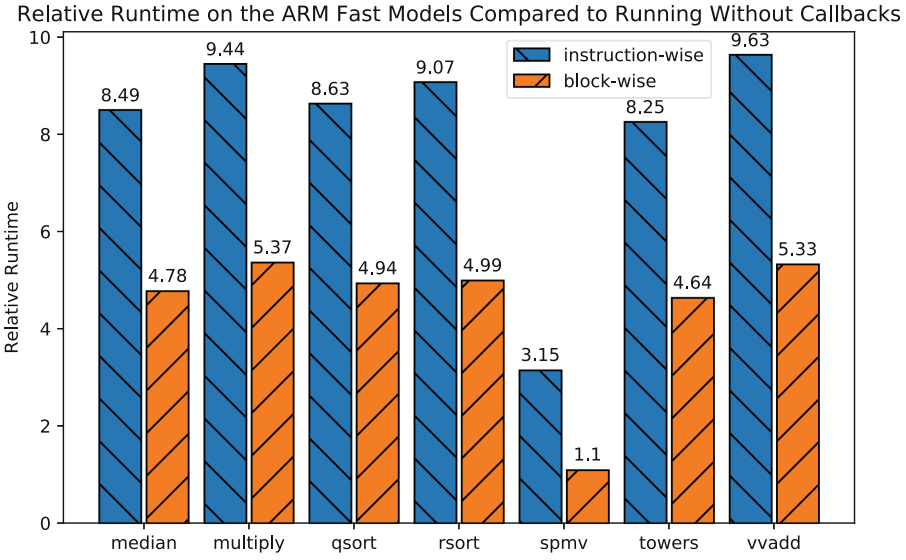

Fig. 5. The measured relative runtime for the different benchmarks when running on the ARM Fast Models. Instruction-wise results were created by executing a callback for each emulated instruction, while block-wise results were created by executing a callback for each basic block.

The particularity of the SPMV benchmarks is the result of the small overall runtime this benchmark exhibits. When increasing the problem size, the relative runtime rises near the other benchmarks. Since this behavior could not be observed for QEMU, an additional overhead at the beginning of the simulation when using the ARM Fast Models within Platform Architect is to be assumed.

Architectural plugins are also provided with the ARM Fast Models. This makes it possible to compare the previous results of the minimal expected overhead with a full pipeline model. Figure 6 shows the relative runtimes of the pipeline model in comparison with the previously measured instruction-wise runtime that can also be seen in Fig. 5. The average relative runtime of the pipeline model amounts to 65.71 meaning that the non-optimizable overhead introduced by calling additional functions amounts in average to approximately 12.31% of the simulation time.

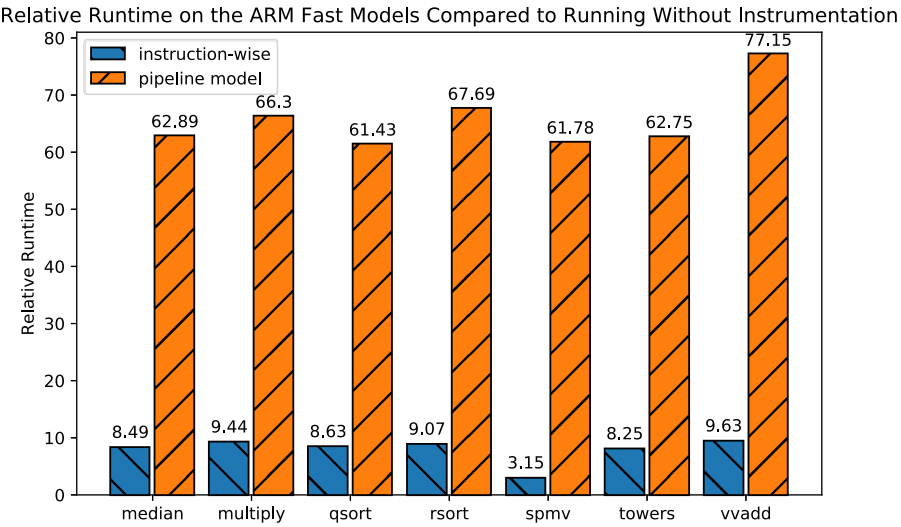

Fig. 6. The measured relative runtime for the different benchmarks when running on the ARM Fast Models. Instruction-wise results were created by executing a callback for each emulated instruction, while pipeline model results were created with the architectural pipeline model enabled.

5 Conclusion

Simulating processor cores and estimating the execution time on the final hardware is important for software development without having the final hardware available. However, since an accurate estimation requires a lot of time, a compromise between the precision and simulation speed must be found. When instrumenting a fast emulator, that uses JIT compilation techniques to speed up its execution, to drive an architectural model, additional overheads have to be taken into account that are not directly introduced by the model of the architecture. Calling anything from within the simulation without even doing anything also requires additional time. This overhead cannot be easily avoided.

In this paper, two emulators, an academic and a commercial one, were investigated regarding the introduced overhead when running callbacks during the simulation. On average, a relative runtime of 2.76 and 8.09 respectively was measured. These numbers are expected to be the minimal overhead introduced by architectural models that are driven by the emulation. Improvements can be achieved by using different approaches like driving a model only each block instead of each instruction. Here, relative runtimes of 2.24 and 4.45 could be measured which is less than the previously mentioned overhead. Thus, it could be shown that certain optimizations to this minimal overhead is still possible but require suitable methods for instrumentation. Finally, comparisons to the overhead of full architectural models were given. The investigated callbacks amount to approximately 10% to 12.31% of the overall runtime.

The findings of this paper can be used in future work as described below. Trying to speed up architectural models has an upper limit in speed which is defined by the emulation and the overhead introduced by instrumenting the simulation. Making fast simulators more accurate by adding an architectural model, similar to the approach as shown here, means that at least the overhead for the instrumentation has to be taken into account and cannot be reduced by optimizing the model.

References

1. Allen, F.E.: Control flow analysis. In: Proceedings of a Symposium on Compiler Optimization. pp. 1–19. ACM, New York, NY, USA (1970). https://doi.org/10.1145/800028.808479
2. ARM Ltd.: Fast Models User Guide
3. ARM Ltd.: Fast models (2019). https://developer.arm.com/products/system-design/fast-models
4. Bellard, F.: QEMU, a fast and portable dynamic translator. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC 2005), USENIX Association, Berkeley, CA, USA (2005)
5. Bringmann, O., et al.: The next generation of virtual prototyping: ultra-fast yet accurate simulation of hw/sw systems. In: 2015 Design, Automation Test in Europe Conference Exhibition (DATE), pp. 1698–1707 (March 2015). https://doi.org/10.7873/DATE.2015.1105
6. Burger, D., Austin, T.M.: The simplescalar tool set, version 2.0. SIGARCH Comput. Archit. News **25**(3), 13–25 (1997). https://doi.org/10.1145/268806.268810
7. Delbergue, G., Burton, M., Konrad, F., Le Gal, B., Jego, C.: 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). TOULOUSE, France (Jan 2016). https://hal.archives-ouvertes.fr/hal-01292317
8. Ding, J., Chang, P., Hsu, W., Chung, Y.: PQEMU: A parallel system emulator based on QEMU. In: 2011 IEEE 17th International Conference on Parallel and Distributed Systems, pp. 276–283 (Dec 2011). https://doi.org/10.1109/ICPADS.2011.102
9. Guo, X., Mullins, R.: Accelerate cycle-level full-system simulation of multi-core risc-v systems with binary translation (2020)
10. Herglotz, C., Seiler, J., Kaup, A., Hendricks, A., Reichenbach, M., Fey, D.: Estimation of non-functional properties for embedded hardware with application to image processing. In: 2015 IEEE International Parallel and Distributed Processing Symposium Workshop, pp. 190–195 (May 2015). https://doi.org/10.1109/IPDPSW.2015.58
11. IEEE Computer Society: IEEE Standard for Standard System C Language Reference Manual, pp. 1666–2011. IEEE Standard (2012)
12. Imperas: Open virtual platforms (2017). http://www.ovpworld.org/
13. Kang, S.H., Yoo, D., Ha, S.: TQSIM: A fast cycle-approximate processor simulator based on QEMU. J. Syst. Archit. **66**, 33–47 (2016). https://doi.org/10.1016/j.sysarc.2016.04.012. http://www.sciencedirect.com/science/article/pii/S1383762116300297

14. Kerschbaumer, C., Hennigan, E., Larsen, P., Brunthaler, S., Franz, M.: Information flow tracking meets just-in-time compilation. ACM Trans. Archit. Code Optim. **10**(4), 38-1–38-25 (2013). https://doi.org/10.1145/2555289.2555295
15. Lee, J., Jang, H., Kim, J.: Rpstacks: fast and accurate processor design space exploration using representative stall-event stacks. In: 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 255–267 (Dec 2014). https://doi.org/10.1109/MICRO.2014.26
16. Luo, Y., Li, Y., Yuan, X., Yin, R.: QSIM: framework for cycle-accurate simulation on out-of-order processors based on QEMU. In: 2012 Second International Conference on Instrumentation, Measurement, Computer, Communication and Control, pp. 1010–1015 (Dec 2012). https://doi.org/10.1109/IMCCC.2012.397
17. Miettinen, A., Hirvisalo, V., Knuuttila, J.: Execution-driven simulation of nonfunctional properties of software. In: Proceedings of European Simulation and Modelling Conference (ESM) (2010)
18. Monton, M., Portero, A., Moreno, M., Martinez, B., Carrabina, J.: Mixed sw/systemc soc emulation framework. In: 2007 IEEE International Symposium on Industrial Electronics, pp. 2338–2341 (June 2007). https://doi.org/10.1109/ISIE.2007.4374971
19. Rosa, F., Ost, L., Reis, R., Sassatelli, G.: Instruction-driven timing CPU model for efficient embedded software development using OVP. In: 2013 IEEE 20th International Conference on Electronics, Circuits, and Systems (ICECS), pp. 855–858 (Dec 2013). https://doi.org/10.1109/ICECS.2013.6815549
20. Sandberg, A., Nikoleris, N., Carlson, T.E., Hagersten, E., Kaxiras, S., Black-Schaffer, D.: Full speed ahead: detailed architectural simulation at near-native speed. In: 2015 IEEE International Symposium on Workload Characterization, pp. 183–192 (Oct 2015). https://doi.org/10.1109/IISWC.2015.29
21. Schreiner, S., Görgen, R., Grüttner, K., Nebel, W.: A quasi-cycle accurate timing model for binary translation based instruction set simulators. In: 2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS), pp. 348–353 (July 2016). https://doi.org/10.1109/SAMOS.2016.7818371
22. Weaver, V.M., McKee, S.A.: Are cycle accurate simulations a waste of time. In: Proceedings of 7th Workshop on Duplicating, Deconstructing, and Debunking, pp. 40–53 (2008)
23. Xilinx: LibSystemCTLM-SoC (2019). https://github.com/Xilinx/libsystemctlm-soc