QEMU-CAS: A Full-System Cycle-Accurate Simulation Framework based on QEMU

Ye Cao
ye.c@rioslab.org
RIOS Lab, Tsinghua University

Zhixuan Xu
sxxuzhixuan@hotmail.com
RIOS Lab, Tsinghua University

Zhangxi Tan
xtan@rioslab.org
RIOS Lab, Tsinghua University

ABSTRACT

Hardware and software co-design is a significant approach in the domain of advanced modern processor design, and can largely help to address the challenges from the early stages of design. Software simulators, developed by software tools, offer enhanced effectiveness and flexibility compared to hardware simulation tools. However, while real-world application benchmarks have gained prominence among architects for evaluating hardware designs, existing software simulators face challenges in effectively modeling complex target systems.

This thesis introduces QEMU-CAS, an innovative full-system software simulation framework that achieves cycle-accurate simulation. QEMU-CAS outperforms existing software simulators by its unique capability to model real-world workloads. To the best of our knowledge, this work represents the first utilization of a performance simulator on a functional full-system platform for RISC-V ISA. Meanwhile, we propose a novel methodology for performance analysis on a dynamic binary translation framework, and a switch-based mechanism for dynamic workload characterization. As a result, QEMU-CAS can serve as an efficient virtual platform for architecture design space exploration and innovation.

KEYWORDS

Computer Architecture, Software Simulation, Modeling

1 INTRODUCTION

Hardware and software co-design, encompassing the integration of software modeling and simulation techniques, holds substantial significance in advanced modern processor design. By utilizing software modeling and simulation, architects can create virtual representations of the system architecture, make easier adjustments, and thoroughly test the design specifications. This iterative process empowers architects to evaluate various architectural choices, analyze their performance implications, and optimize the system performance and design efficiency.

To measure the capabilities and performance of hardware, architects develop various benchmarks as standards[21, 7, 9]. So far, traditional static benchmarks are facing increasing challenges in remaining relevant and up-to-date with emerging hardware technologies. This is due to limitations in various aspects, including a lack of representativeness of the comprehensive performance picture, limited scalability, and inadequate coverage of emerging workloads. As a result, the rapid evolution of hardware architectures and system requirements demands a more dynamic approach to benchmarking, which can capture real-world conditions by incorporating diverse workloads.

However, modeling real-world applications via software presents several challenges. Performance simulators face limitations in terms of flexibility and scalability when simulating large workloads. This leads to a low execution speed[19] and a lack of capabilities to scale the simulated system models. Besides, it is also challenging to achieve accurate and efficient simulation of IO device modeling. The diverse nature of specifications, protocols, and interfaces further amplifies the efforts and difficulties.

In contrast, functional simulators offer higher performance and often enable full-system simulations. However, their design focus on functional modeling limits their effectiveness in conducting detailed performance analysis. These approaches may not provide architects and researchers with sufficient information to optimize and refine their designs in an effective manner.

This thesis presents QEMU-CAS, a novel full-system cycle-accurate software simulation framework. QEMU-CAS integrates a cycle-accurate CPU model with QEMU[16, 4], a popular machine emulator with a dynamic binary translation framework, to enable efficient modeling and simulation for complete systems. Notably, QEMU-CAS surpasses existing software simulators by offering capabilities to model full-fledged Linux environments with modern IO peripherals. To the best of our knowledge, QEMU-CAS is the first work that introduces cycle-accurate performance modeling on top of a functional full-system platform for RISC-V ISA[8, 15]. We introduce a new methodology to gather performance statistics for architecture emulators executing instruction streams via dynamic binary translation frameworks. It is a valuable tool for exploring the architecture design space and holds significant potential for future extensions and advancements.

This thesis makes the following contributions:

(1) A simulation framework capable of modeling superscalar out-of-order RISC-V processors.
(2) A novel methodology for full-system modeling and simulations on RISC-V ISA from the adoption of QEMU.
(3) A dynamic switch-based mechanism to characterize target workloads and make performance analysis on a dynamic binary translation framework.

2 RELATED WORK

We can categorize the software simulators by the abstraction level of their simulation. Functional simulators focus on high performance and correct functionalities during the simulation and simulate less architecture information. On the other hand, timing simulators, also referred to as performance simulators, are capable of detailed performance analysis, which could achieve accuracy at the timing level, and can simulate the target specification more precisely.

According to detailed timing implementations, architects usually categorize the simulators into event-driven and execution-driven. Execution-driven simulators simulate every individual instruction in order and update the architecture states accordingly. In contrast,

event-driven simulators execute instructions in response to hardware events. With less information to collect, they are usually faster than execution-driven simulators, yet achieve lower accuracy.

Mainstream performance simulators have various designs and thus lead to distinct features. PTLSim[22] enables flexible configurations and co-sim technology but only supports X86 ISA. SimpleScalar[2] is capable of modeling many CPU architectures, while the official maintenance of its framework updates slowly. Sniper[6] and Gem5[5, 10] are sophisticated event-driven full-system simulators and could achieve good simulation performance. However, they cannot run cycle-accurate simulations, resulting in inconvenience for RTL debugging. RISC-V Foundation proposes Sparta[20], a simulation toolkit for both functional and performance simulation. However, the provided implementation is a trace-driven CPU model[18], and it has challenges handling complex systems.

On the other hand, functional simulators usually have enhanced abilities in performance and functionalities. Simics[12] enables architects to run a virtual system and do full-system simulation. Dromajo[1] and RISC-V Spike[17] are popular simulators for designs of RISC-V processors. QEMU[16], a system simulator, can also achieve the functional simulation, which uses binary translation[3] techniques for fast emulation.

Researchers have demonstrated significant interest in QEMU, primarily driven by its unique design and the potential for integration with existing simulators. Yan Luo etc.[11] bring a model using SimpleScalar as backends and use QEMU as the functional frontend of the simulator. Avadh Patel etc. develop MARSS, an X86-64 ISS[14] which combines PTLSim and QEMU and can change CPU models at runtime. However, these designs do not get rid of the limitations such as slow maintenance or ISA dependence, which come from existing simulators they use.

3 DESIGN METHODOLOGY

This section introduces the methodology of QEMU-CAS. Our framework aims to enable architects and researchers in full-system simulation and analysis towards real workloads and provide an efficient tool for architecture design testing and exploration.

3.1 Component

QEMU-CAS uses an execution-driven model on CPU simulation and uses the main clock conversion of hardware as the main timeline of the simulator. This approach is closer to the implementation on a real board and is more intuitive for performance analysis. At the same time, QEMU-CAS uses event-driven models for the interactions of IO devices when simulating SoC.

Components are the basic units of emitting, receiving, and processing events. QEMU-CAS uses this abstraction to simulate both CPU and IO. For CPU modeling, there are 2 types of resources during the abstraction:

1) Stages. Stages are the highest layer of the CPU model and maintain the timing and pipeline structure of the CPU. In the main execution loop, the simulator executes each stage sequentially according to a preset order.

The execution-driven model gives QEMU-CAS high flexibility when designing pipeline stages. The main loop, rather than the stages themselves, drives the CPU simulation. In contrast, the event-driven model such as the Gem5 O3 model uses a large event queue for the simulation. If the simulator does not update a certain stage during the execution, it will not add this stage to the event queue. This may provide obstacles to collecting performance statistics.

2) Function Components. Function Components (FCs) are responsible for detailed functionalities, including both data queues and function units. Some function units have complex and pipelined structures and may take multiple cycles to handle instructions. In these cases, FCs include internal registers to represent timing logic.

3.2 Design of QEMU-CAS

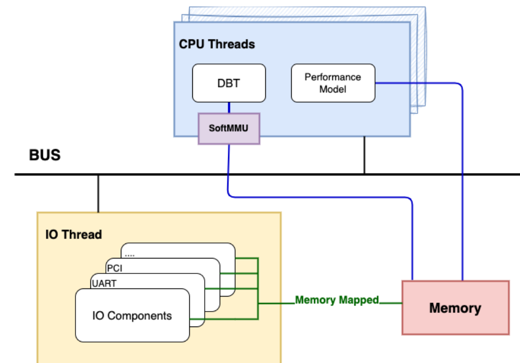

Figure 1: QEMU-CAS Simulation Infrastructure

Figure 1 shows the basic infrastructure of the QEMU-CAS framework. It adopts from QEMU platform and incorporates IO devices as separate components which can send and receive signals over the bus. The multi-thread architecture brings an efficient workload simulation, as IO devices and CPU models could run simultaneously on different threads.

Timing and functionality are not as relevant in the SoC simulation as they are in CPU models. System timing largely depends on a global clock ticking at a constant frequency. Meanwhile, IO components connect each other with wires, so their communications can be independent of clock ticks. Therefore, an event-driven model can efficiently simulate the SoC behaviors, where a main loop runs continuously and handles the IO events of various devices. In summary, QEMU-CAS uses hybrid-driven approaches on its whole framework, which drives the full-system simulation with events and executes instructions in execution-driven models.

Apart from independent threads representing CPU and IO, QEMU-CAS also includes a stand-alone memory model. Since QEMU-CAS decouples the memory from IO and CPU threads, customizing the memory model becomes easier for architects. Architects can make customization on top of the existing model, and achieve more complex functionalities.

3.3 CPU-IO Interface

In this section, we propose the general CPU-IO interfaces of QEMU-CAS, including memory and Interrupt-Request (IRQ) interfaces. With these interfaces, QEMU-CAS provides an efficient approach

to facilitating communication between a cycle-accurate CPU model and an emulated SoC.

3.3.1 Memory Interface. We take advantage of QEMU's memory model, a well-established and tested framework for managing memory and IO operations. When the CPU model requires data in the Load/Store Unit, the CPU accesses the memory through an emulated bus and seeks results. We decouple the timing behaviors of the memory model from its functionalities. Architects have free space to specify their desired delay before or after accessing the memory.

3.3.2 IRQ Interface. In addition to memory access, IO devices can also contact the CPU by sending IRQs. An interrupt is a signal sent by an IO device to the CPU to request its attention. When an IO device needs to communicate with the CPU, it can assert an interrupt signal, which causes the CPU to stop executing its current task and handle the interrupt.

For functional emulation, QEMU executes each instruction in one cycle. So the whole execution loop could stop immediately when receiving an IRQ. But for cycle-accurate simulation, an interrupt may need several cycles before it commits. Therefore, it is important to maintain the synchronization between cycle-level instruction execution and functional IO emulation.

QEMU-CAS uses a vector to record the essential architecture information. Each processor owns a separate vector entry, including an IRQ code (if exists), the current privilege level of the CPU, and CSR states. When an IO device sends an IRQ to the CPU, it updates the IRQ code in the associated entry. The privilege level bit is only writable to the CPU, while Interrupt Controllers can read this bit and get information about the CPU's current privilege mode.

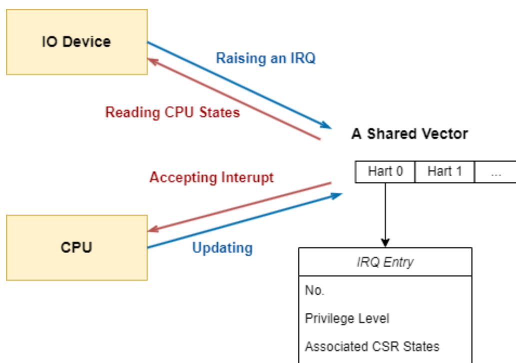

Figure 2: CPU and IO exchange IRQ through an IRQ Vector

3.4 Assistance for Architecture Design Space Exploration

QEMU-CAS allows for customizing core topology in the configuration file, which largely surpasses the capability of mainstream simulators and helps architects to test and tune their hardware architectures.

Architects can customize not only the hardware parameters but also the pipeline stages. They could preset the stage dependencies in the configuration file before execution. The simulator executes each stage sequentially, following the stage sequence in the preset configurations.

Additionally, QEMU-CAS allows for customizing the FCs involved in data operations, such as issue queues and function units. Architects could set them as parameters for other components in the configuration file when they need to adjust the system layout. Taking an example in designing the Dispatch stage, architects can freely combine issue queues, functional units, and related ports according to their needs only by modifying the configurations.

In particular, QEMU-CAS decouples the ISA model from CPU models. Only in certain stages such as Decode, core models call the ISA model for associate functions. Therefore, it is easy to make adjustments to ISA specifications, such as adding new instructions or extensions.

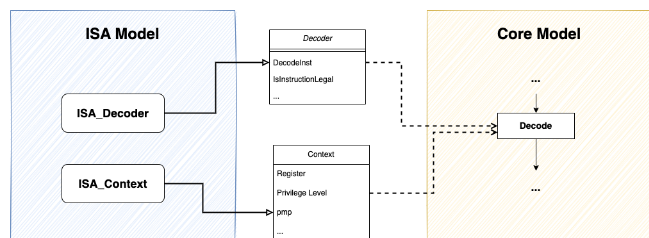

Figure 3: The ISA model is independent of the core model.

4 SIMULATING MULTI-CORE TARGET SYSTEMS

As the prevalence of multi-core processors continues to grow, simulation frameworks that do not incorporate support for multi-core environments may offer an inadequate understanding of system performance. Hence, it is crucial for simulators to study the behaviors of multi-core systems. By achieving the multi-thread architecture, QEMU-CAS allows for efficient parallel simulation of the workload. In handling multiple cores, the simulator generates a distinct instance for each hart, which runs execution in a separate thread of the host machine.

A key issue in parallel simulation is dealing with the trade-offs between performance and accuracy. For extreme cases, a cycle-accurate simulator could do synchronization every cycle, which is surely the most accurate way but will have countable performance bound. The quanta-based approach is one of the most popular solutions for parallel simulators. During the simulation, each thread owns a quantum of simulated cycles. When running off the quanta, threads synchronize with each other. There are a lot of popular simulators using this idea, such as Graphite[13] and Sniper[6].

QEMU-CAS uses a pre-configured quantum to achieve the simulation of multi-core systems. When the quantum reaches zero, the simulation stops and the simulator starts the synchronization. Due to the execution-driven structure, it is efficient to stop the simulation at the synchronization barriers. In contrast, an event-driven simulator will have much larger overheads since it processes events in transaction.

5 DYNAMIC MODEL SWITCHING

In this section, we present a switch-based mechanism to characterize the workloads, which enables the simulator to switch seamlessly and lively between a functional DBT and a cycle-accurate CPU model. This mechanism is valuable for the testing and debugging purposes of hardware development, and particularly useful when working with complex systems that are difficult to debug or diagnose using traditional methods.

5.1 Checkpoint Strategy

When the simulator receives a switching command, the current core should immediately cease its execution. However, the response may have a delay in real implementations, which may affect the correctness of performance behavior. Meanwhile, it is also impractical to largely sacrifice the original performance to create checkpoints. To balance this trade-off, we propose different solutions concerning different architectures of the DBT and the performance simulator.

5.1.1 Checkpointing of DBT. QEMU DBT adopts translation blocks (TBs) as its base unit of code generation. It batches a contiguous sequence of instructions in a TB to achieve a more efficient translation. Since no speculative operations cross TBs, it is easy to stop the execution at the beginning or end of a TB. To facilitate model switching, we introduce a new exception code with the lowest priority, representing model switching.

When DBT receives a switch request and there are no other exceptions, the exception handler accepts the new exception and stops execution as usual. QEMU-CAS then handles the corresponding synchronizations after receiving the return value. In the presence of other exceptions, the exception handler prioritizes the higher-priority exception and resumes execution after handling it. This process repeats until DBT handles all other exceptions, and then DBT stops executing to perform the necessary synchronization.

5.1.2 Checkpointing in Performance Model. The performance model precisely tells the timing behaviors of the core. It is very common for an instruction to stay in the pipeline for multiple cycles before it commits. When a switching signal arrives, the whole pipeline should run to a certain state that can synchronize with DBT which owns simulation at instruction-level accuracy.

In the pipeline frontend including Fetch and Decode stages, the simulator decodes the instructions into ISA information. Before the pipeline sends frontend instructions to issue queues and assigns them to function units, these instructions would not affect the architecture states. But after they arrive at the backend, the instructions do change some architecture states. The determination of the impact of an instruction occurs when the instruction finally commits. Otherwise, the simulator will recover the changes of architecture states.

However, a switching command causes the termination of the whole pipeline, resulting in information loss of uncommitted instructions. Figure 4 indicates the scenario when the simulator receives switching requests. When the simulator suddenly stops and switches to DBT, the pipeline loses all uncommitted instructions that have started executing.

To ensure the correctness of the simulation before and after switching, the simulator should handle timing-associated processes in transaction. The performance model utilizes a switching strategy that activates only after meeting several prerequisites.

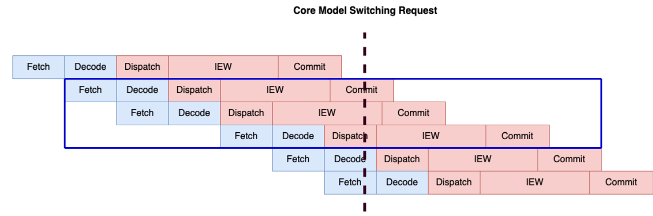

Figure 4: When receiving a switching request, the pipeline loses all instructions in the square.

- None instructions are at the backend.
- At least one fetched instruction in the frontend
- No pending interrupts in pipeline

Once meeting these prerequisites, the backend is clear. The simulator can create checkpoints according to the oldest PC in the pipeline. This ensures that the pipeline has either committed or flushed all instructions newer than the oldest PC and thereby accurately synchronizes all CPU states at this tick to the DBT.

5.2 Time Dilation

In an SoC, a global clock ticks at a constant frequency. Simulating the global clock is a crucial part of the cycle-accurate simulation. Inaccurate clock simulation can lead to timing errors in the simulation, which can have a significant impact on the overall accuracy of the simulation results.

For a specific simulator, however, the simulated time could not always keep pace with the real-world time. This also refers to time dilation, a rate change of the simulated time.

5.2.1 Timer. In QEMU-CAS, we simulate a global clock linearly related to host OS time. The timer is a vector including $base_freq$ and $delta$, the relative frequency of the timer, and the offset for time calculation. Equation 1 presents the timestamp of the virtual world, whose unit is one second.

$$TS_{virtual}(TS_{rt}) = TS_{rt} * base_freq + delta \tag{1}$$

Once the simulator changes time, the time of the virtual clock and that of the real-world clock are no longer simply proportional. Therefore, we introduce $delta$ as an offset. Whenever the simulator rewrites time, it sets $delta$ as the difference between the new value and the original value.

5.2.2 Clock Alignment in Core Switching. A read-only Flattened Device Tree (FDT) records the parameters of each hardware device. The bootloader and Linux kernel load the FDT when the machine starts. Configurations of CPUs need to be the same when multiple models both simulate the same CPU. Otherwise, several mistakes will occur.

In the simulation, CPU frequency refers to the simulation speed of the CPU model. It is the ceiling of the model's performance and is difficult to get further improved. In case multiple models simulate a single CPU, the performance CPU model is unlikely to run as fast as DBT. This could be a crucial issue that results in errors in Linux

Table 1: Parameters of core models

Item	Configuration
Fetch Width	8
ROB Entry Size	32
Load/Store Queue Entry	16
TLB Entry	8
BTB Size	1024
RAS Size	16
Tournament - Local Predictor Size	1024
Tournament - Local History Entry Bits	10
Tournament - Global History Entry Bits	12
Physical Register File	128

time slicing. To provide a solution, we propose a dilation strategy to keep the CPU frequency relatively constant.

Equation 1 represents the time when the simulator switches the core model. We can set up a system of equations, and get the modified $delta$ when keeping the target CPU frequency unchanged:

$$\begin{aligned} TS_{virtual}(TS_{rt}) &= TS_{rt} * base_freq_{DBT} + delta_{DBT} \\ TS_{virtual}(TS_{rt}) &= TS_{rt} * base_freq_{model} + delta_{model} \end{aligned} \tag{2}$$

$$delta_{model} = TS_{rt} * (base_freq_{DBT} - base_freq_{model}) + delta_{DBT} \tag{3}$$

Clock change occurs during the switch. At that time, we use a global mutex lock to lock every IO device except the timer. After clock changes, guest OS will run at a slower speed by observation from the real world. However, for the simulated workload, all timing behaviors remain unchanged.

6 EVALUATION

In this section, we conduct several experiments to evaluate QEMU-CAS in different aspects.

We use an execution-driven performance model of a superscalar out-of-order RISC-V CPU, including performance simulations of TLB, BPU, and other components. Since the detailed implementation of simulating each CPU component is not our focus in this paper, we just list the configurations of the CPU model used in these experiments in Table 1.

With the integration of a cycle-accurate model, QEMU-CAS is able to capture precise architecture behaviors. We test all the case studies based on a RISC-V Fedora Rawhide Linux, and collect the architectural statistics in a performance monitoring unit (PMU).

6.1 Simulator Performance

Table 2 shows the performance of different core models in QEMU-CAS. Almost all performance bottlenecks come from the performance model as QEMU DBT runs about 10^3 times faster. In contrast, removing the BPU from the performance model does not affect the performance bottleneck. Changing the internal system design causes about 18% slower than the original model, far less when compared to DBT performances.

Table 2: Performance of DBT and core model

Simulator	Time per cycle(μs)
QEMU DBT	0.241
Performance Model	249.024
Performance Model, no BPU	294.363

Table 3: Ping Test Script

ping_test.sh
./prof_tool begin ping -c *n url* > *output_file_path* ./prof_tool terminate

6.2 Case Study: Linux Ping

In this experiment, we use Linux Ping as a benchmark to test our capability to interact with IO devices. Table 3 shows the script we use on a Fedora Linux. We sample the execution per 0.1 million cycles and plot several critical performance metrics, including IPC, DTLB hit rate, and BPU accuracy. Figure 5 shows the result diagrams.

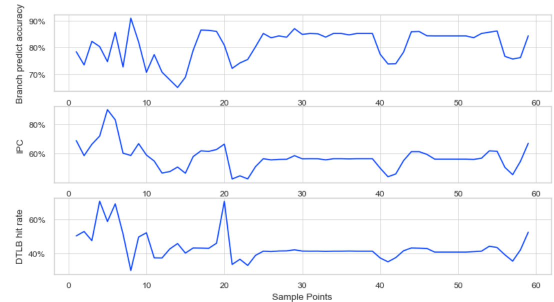

Figure 5: PMU Result of Executing Ping.

We can see from the diagrams that the curves have 3 relatively large fluctuations, nicely corresponding to the 3 packet exchanging process. Upon the execution and the resulting diagram, QEMU-CAS has demonstrated its ability to accurately simulate the core behaviors of network devices as well as the support for simulated block devices.

The successful execution of ping signifies that our simulator can effectively simulate popular IO devices. It can be a valuable tool in the development and testing process of various applications and systems that rely on these functionalities.

6.3 Case Study: Socket Communication

In this experiment, we aim to investigate the hardware behavior in a socket network. To accomplish this, we build a server/client framework through a socket protocol. Figure 6 shows its basic structure. We run the server on the simulated OS, and the client on another process in the host machine. During the experiment, the client uses a socket to send data to the server, and the server

replies after processing the data. This will repeat several times, and we aim to investigate the behavior of the server's data processing.

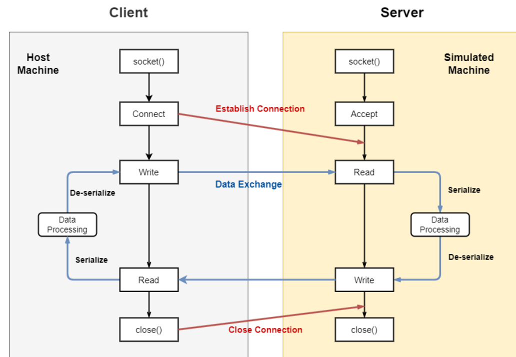

Figure 6: Structure of Server/Client network

During the experiment, we especially focus on the behavior of data processing, not counting the procedures that occur in socket communication. A good way is to use profiling to only trace the desired program segments. The traditional method used by existing simulators is to build checkpoints to locate the start points of tracing and then generate snapshots. However, this will face challenges in capturing an accurate performance picture.

In the interactive scenarios, static snapshots may not capture the whole state, since the client works on another process or even another machine. Furthermore, snapshots need to contain the states of the entire machine, including memory, CPU states, and IO devices, which can be cumbersome when the scale turns large.

QEMU-CAS overcomes these challenges with its dynamic switch-based approaches. It enables the simulator to achieve a seamless and live change of the CPU model, allowing for easy tracing of target program segments.

To capture the hardware behavior of the server's data processing phase, we use DBT to quickly execute the socket connection parts. We switch to a high-accuracy CPU model when the server processes data to trace the hardware behavior completely and precisely, and then switch back to handle the socket.

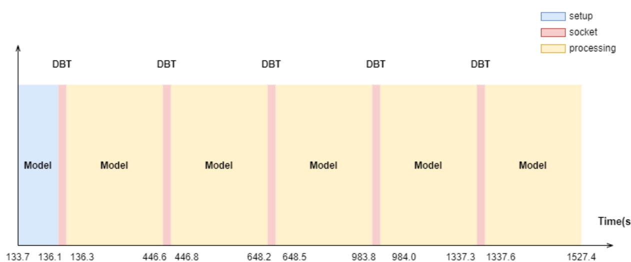

Figure 7: Time lapse graph of execution. The x-axis is the time spent during sampling. The blue/yellow/red blocks represent setup, socket connecting, and data processing.

We create a time-lapse graph in Figure 7. The results indicate that the performance model takes a significant portion of time, while QEMU takes only less than 1% time. These findings demonstrate the efficiency of QEMU in simulation. Moreover, we draw PMU statistics during the execution. The result diagrams in Figures 8 demonstrate the correspondence to Figures 7.

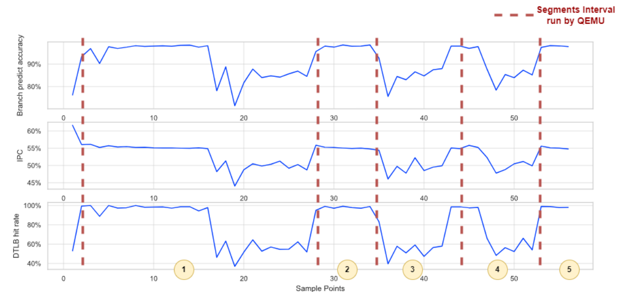

Figure 8: The interval separated by the red line represents the analysis result of PMU. We label the phases of arithmetic processing on X-axis.

For arithmetic phases, the trend of curves is not quite the same in different periods. We could find that in Phase 2 and Phase 5, the curves are smooth, while the other 3 phases have large fluctuations. This is mainly because of interrupts and exceptions generated by the OS either randomly or periodically.

This provides a practical example to demonstrate the capability of locating the target program segments with the help of hybrid simulation, while static approaches will have challenges. In practice, QEMU-CAS provides architects high flexibility on debugging and testing, allowing for detailed analysis as well as a high degree of suitability for specific needs.

7 CONCLUSION

This work introduces a software simulator capable of full-system simulation. Our simulator exhibits great capabilities by well modeling complex target hardware systems with high accuracy. Furthermore, the paper proposes a dynamic and agile switch-based mechanism to characterize large target workloads. It allows architects and researchers substantial time and effort savings and can largely increase the efficiency of the simulation of large-scale workloads.

This paper also conducts several experiments on real-world workloads. The evaluation results demonstrate the value and capability of QEMU-CAS for assisting the architecture design of RISC-V. Our ongoing effort is focusing on the implementation of more advanced systems, to better provide the simulation and optimization for real-world scenarios.

REFERENCES

[1] Chips Alliance. [n. d.] Dromajo: an open-source risc-v processor verification framework. https://github.com/chipsalliance/dromajo. Accessed: April 15, 2023. ().

[2] Todd Austin, Eric Larson, and Dan Ernst. 2002. Simplescalar: an infrastructure for computer system modeling. *Computer*, 35, 2, 59–67.

[3] Fabrice Bellard. 2005. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*. Vol. 41. Califor-nia, USA, 46.

[4] XiaoXiao Bian. 2017. Implement a virtual development platform based on qemu. In *2017 International Conference on Green Informatics (ICGI)*. IEEE, 93–97.

[5] Nathan Binkert et al. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39, 2, 1–7.
[6] Trevor E Carlson, Wim Heirman, and Lieven Eeckhout. 2011. Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–12.
[7] EEMBC. [n. d.] Coremark: a simple, yet sophisticated, benchmark for embedded processors. https://github.com/eembc/coremark. Accessed on April 15, 2023. ().
[8] RISC-V Foundation. [n. d.] Risc-v. https://riscv.org. Accessed: April 15, 2023. ().
[9] John L Henning. 2006. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34, 4, 1–17.
[10] Jason Lowe-Power et al. 2020. The gem5 simulator: version 20.0+. *arXiv preprint arXiv:2007.03152*.
[11] Yan Luo, Ying Li, Xinyu Yuan, and Rong Yin. 2012. 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*. IEEE, 1010–1015.
[12] Peter S Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. 2002. Simics: a full system simulation platform. *Computer*, 35, 2, 50–58.
[13] Jason E Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. 2010. Graphite: a distributed parallel simulator for multicores. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. IEEE, 1–12.
[14] Avadh Patel, Furat Afram, and Kanad Ghose. 2011. Marss-x86: a qemu-based micro-architectural and systems simulator for x86 multicore processors. In *1st International Qemu Users' Forum*. Citeseer, 29–30.
[15] David A Patterson. 1985. Reduced instruction set computers. *Communications of the ACM*, 28, 1, 8–21.
[16] [n. d.] Qemu. https://www.qemu.org/. Accessed: April 15, 2023. ().
[17] RISC-V Foundation. [n. d.] RISC-V ISA Simulator (Spike). https://github.com/riscv-software-src/riscv-isa-sim. Accessed: April 29, 2023. ().
[18] RISC-V Software Source. [n. d.] RISC-V Performance Modeling Framework. https://github.com/riscv-software-src/riscv-perf-model. Accessed: April 30, 2023. ().
[19] Wilson Snyder and Dan Gisselquist. [n. d.] Verilator: open source verilog simulator. https://www.veripool.org/papers/Verilator_Modeling_UMass2017b_pres.pdf. Accessed on 18 May 2023. ().
[20] [n. d.] The Sparta Modeling Framework. https://sparcians.github.io/map/index.html. Accessed: April 30, 2023. ().
[21] Reinhold P Weicker. 1984. Dhrystone: a synthetic systems programming benchmark. *Communications of the ACM*, 27, 10, 1013–1030.
[22] Matt T Yourst. 2007. Ptlsim: a cycle accurate full system x86-64 microarchitectural simulator. In *2007 IEEE International Symposium on Performance Analysis of Systems & Software*. IEEE, 23–34.